Proceedings of

Formal Methods in Computer Aided Design

FMCAD 2006

Proceedings of

Formal Methods in Computer Aided Design

12-16 November 2006, San Jose, California, USA

Los Alamitos, California

Washington • Tokyo

IEEE Computer Society Order Number P2707
ISBN 0-7695-2707-8
ISBN 978-0-7695-2707-9
Library of Congress Number 2006935247

Additional copies may be ordered from:

IEEE Computer Society
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1314
Tel: + 1 800 272 6657
Fax: + 1 714 821 4641
http://computer.org/cspress
csbooks@computer.org

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
Tel: + 1 732 981 0060
Fax: + 1 732 981 9667
http://shop.ieee.org/store/
customer-service@ieee.org

IEEE Computer Society
Asia/Pacific Office
Watanabe Bldg., 1-4-2
Minami-Aoyama
Minato-ku, Tokyo 107-0062
JAPAN
Tel: + 81 3 3408 3118
Fax: + 81 3 3408 3553
tokyo.ofc@computer.org

Individual paper REPRINTS may be ordered at: <reprints@computer.org>

Editorial production by Bob Werner
Cover art production by Joe Daigle/Studio Productions
Printed in the United States of America by Applied Digital Imaging

IEEE Computer Society
Conference Publishing Services
http://www.computer.org/proceedings/

Table of Contents: FMCAD 2006

Formal Methods in Computer Aided Design

Model Checking

Automata Theoretic Methods

Theorem Proving

Testing and Verification Applications

Organizing Committee

Chairs
Aarti Gupta, NEC Laboratories America, USA
Panagiotis Manolios, Georgia Institute of Technology, USA

Local Arrangements
Jeremy Levitt, Mentor Graphics, USA
Vigyan Singhal, Oski Technology, USA

Panels
Andreas Kuehlmann, Cadence Labs, USA

Tutorials
Leonardo de Moura, SRI International, USA

Webmasters
Sudarshan Srinivasan, Georgia Institute of Technology, USA
Daron Vroon, Georgia Institute of Technology, USA

Workshops
Ganesh Gopalakrishnan, University of Utah, USA

Program Committee
- Clark Barrett, New York University, USA
- Jason Baumgartner, IBM Corporation, USA
- Valeria Bertacco, University of Michigan, USA
- Dominique Borrione, Grenoble University, France
- Supratik Chakraborty, Indian Institute of Technology Bombay, India
- Alessandro Cimatti, Istituto per la Ricerca Scientifica e Tecnologica, Italy
- Edmund M. Clarke, Carnegie Mellon University, USA
- Leonardo de Moura, SRI International, USA
- Rolf Drechsler, University of Bremen, Germany
- Malay K. Ganai, NEC Laboratories America, USA
- Ganesh Gopalakrishnan, University of Utah, USA
- Susanne Graf, VERIMAG, France
- Orna Grumberg, Technion - Israel Institute of Technology, Israel
- Aarti Gupta, NEC Laboratories America, USA
- Alan J. Hu, University of British Columbia, Canada
- Warren Hunt, University of Texas, USA
- Andreas Kuehlmann, Cadence Laboratories, USA
- Panagiotis Manolios, Georgia Institute of Technology, USA
- Andy Martin, IBM Research Division, USA
- Ken McMillan, Cadence Labs, USA
- John O'Leary, Intel Corp., USA
- Wolfgang Paul, Saarland University, Germany
- Carl Pixley, Synopsys Inc., USA
- Amir Pnueli, NYU, USA
- Natarajan Shankar, SRI International, USA
- Mary Sheeran, Chalmers University of Technology, Sweden
- Eli Singerman, Intel Corp., Israel
- Vigyan Singhal, Oski Technology, Inc., USA
- Anna Slobodova, Intel Corp., USA
- Fabio Somenzi, University of Colorado at Boulder, USA
- Richard Trefler, University of Waterloo, Canada
- Matthew Wilding, Rockwell Collins Inc., USA
- Yaron Wolfsthal, IBM, Israel

Referees

Johan Alfredsson
Zaher Andraus
Tamarah Arons
Mona Attariyan
Ittai Balaban
Felice Balarin
Sharon Bar-Ner
Constantinos Bartzis
Jason Baumgartner
Shoham Ben-David
Sergey Berezin
Ritwik Bhattacharya
Jesse Bingham
Dominique Borrione
Marius Bozga
Marco Bozzano
Karl Brace
Roberto Bruttomesso
Paul Caspi
Supratik Chakraborty
Ching-Tsun Chou
Jared Davis
Rolf Drechsler
Bruno Dutertre
Ali El-Zein
Emmanuelle Encrenaz
Eitan Farchi
Goerschwin Fey
Alon Flaisher
Anders Franzén
Oded Fuhrman
Eric Gascard
Malay K Ganai
Amit Goel
David Greve
Pascal Gribomont
Alberto Griggio
Daniel Grosse
Aarti Gupta
Amit Gupta
Ziyad Hanna
David Hardin
John Harrison
Tamir Heyman
Håkan Hjort
Sonjong Hwang
Beth Isaksen
Sumit Jha
Robert Kanzelman
Zurab Khasidashvili
Kalyanasundaram Krishnamani
Sava Krstic
Robert Krug
Ulrich Kuehne
Hillel Kugler
Flavio Lerda
Mark Liffiton
Hanbing Lui
Stephen Magill
Andy Martin
Arie Matsliah
Vaibhav Mehta
Steve Miller
Maher Mneimneh
Hari Mony
In-Ho Moon
Katell Morin-Allory
Mark Moulin
Leonardo de Moura
Rotem Oshman
Robert Palmer
Viresh Paruthi
Nir Piterman
Carl Pixley
Stephen Plaza
David Rager
Sandip Ray
Erik Reeber
Marco Roveri
Hassen Saidi
Roberto Sebastiani
Ohad Shacham
Narendra Shenoy
Sharon Shoham
Smitha Shyam
Eli Singerman
Nishant Sinha
Anna Slobodova
Serita Nelesen
Sol Swords
Muralidhar Talupur
Daijue Tang
Andrei Tchaltsev
Andreas Tiemeyer
Daniel Tille
Stefano Tonetta
Stavros Tripakis
Rachel Tzoref
Tatyana Veksler
Ilya Wagner
Thomas Wahl
Chao Wang
Mike Whalen
Yaron Wolfsthal
Jessie Xu
Avi Yadgar
Yu Yang
Karen Yorav
Ganna Zaks
Fadi Zaraket
Qiang Zhang
Haifeng Zhu
Lenore Zuck

Enabling Large-Scale Pervasive Logic Verification through Multi-Algorithmic Formal Reasoning

Tilman Glökler[1] Jason Baumgartner[2] Devi Shanmugam[2] Rick Seigler[2]
Gary Van Huben[2] Barinjato Ramanandray[1] Hari Mony[2] Paul Roessler[2]

[1]IBM Deutschland Entwicklung GmbH [2]IBM Systems & Technology Group

***Abstract*— *Pervasive Logic* is a broad term applied to the variety of logic present in hardware designs, yet not a part of their primary functionality. Examples of pervasive logic include initialization and self-test logic. Because pervasive logic is intertwined with the functionality of chips, the verification of such logic tends to require very deep sequential analysis of very large slices of the design. For this reason, pervasive logic verification has hitherto been a task for which formal algorithms were not considered applicable.

In this paper, we discuss several pervasive logic verification tasks for which we have found the proper combination of algorithms to enable formal analysis. We describe the nature of these verification tasks, and the testbenches used in the verification process. We furthermore discuss the types of algorithms needed to solve these verification tasks, and the type of tuning we performed on these algorithms to enable this analysis.

I. Introduction

High-performance digital designs such as the Cell Processor [1] and Pentium [2] generally have extremely aggressive clock frequency goals, which requires manual optimization of the logic, circuits, and even the layout of critical portions of the design. Such manual optimization amplifies the risk of functional design errors, which must be identified and rectified as early as possible in the design flow to avoid costly design and manufacturing iterations.

Apart from the primary functionality of a design such as *arithmetic* or *instruction decode* logic, virtually all modern designs include *Pervasive Logic* (PL) [3]. The PL includes logic for controlling the power-on-reset initialization sequence, for security features to boot only trusted software, and for enabling manufacturing test capabilities such as built-in self-test for logic and memory arrays. It also includes debug functionality for trace logic analysis and to enable access to internal latches and arrays in the chip.

The PL of a design must be customized to the needs of the specific design microarchitecture and circuit technology, thus, it is often entirely manually designed. Furthermore, mainly for timing reasons, this custom PL must often be manually integrated into the functional logic itself, and the two are tightly intertwined. For example, the *scan chains*, which serially connect many of the latches in the chip for initialization, tracing, and test purposes, are often manually connected within the design HDL itself. This greatly contributes to the risk of implementation errors within that pervasive logic or even the functional logic itself. While the goal of functional verification is to validate that the design correctly implements its specification despite the intertwined PL, the goal of *Pervasive Verification* (PV) is to further guarantee that the PL works as intended. In many ways, the correctness of the PL is *more* critical than the correctness of the functional logic, since an error in the latter may well render a costly fabrication of a chip entirely unusable or untestable, whereas a functional logic error at least enables the analysis of other aspects of the chip and may often have a software or hardware workaround.

A. Pervasive Verification Tasks

Most pervasive verification tasks can be subdivided into validating the following categories of PL.

a) POR, Security, and eFuses: Power-on reset (POR) refers to the procedure of initializing or *booting* the chip after enabling its voltage supply, and is responsible for ensuring that the chip is brought to a consistent initial state to ensure proper functional behavior when control is handed over to software execution. During POR, almost all other pervasive functionalities are needed such as initialization of latches and arrays in the proper order, sensing of the security keys in the electric fuses (eFuses) [4], processing security information to ensure that only trusted software can be run on the system [5], bringing up the physical Input / Output interface, enabling runtime error analysis in debug mode, etc.

b) External Debug Interfaces: External debug interfaces such as JTAG [6], SPI [7], and I^2C [8] are used to debug logic errors and analyze manufacturing problems in a chip. These interfaces have access to debug registers which control various functions in the chip such as clocking and Input / Output setup. These serial interfaces are also used to access the scan chains and, thus, enable access to nearly all latches and arrays in the chip. Special features for supporting the reliability, availability and serviceability of the chip allow handling of recoverable or unrecoverable errors (e.g., uncorrectable memory errors), and provide an interface to system software. Using these mechanisms, error recovery can be performed on-the-fly by system software.

c) Debug and Built-In Self-Test Logic: The scan chains in the chip are used for various purposes including initialization during POR, reading latch values onto off-chip interfaces, setting the chip into specific configuration modes, and built-in self-test.

The trace bus and trace logic analyzer functionality is used to monitor thousands of chip-internal signals in real-time, non-intrusively, while the chip is functionally running. T

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

is configurable either by a scan operation or other dedicated registers, and can select among various internal signals to be analyzed by an on-chip evaluation circuit. This functionality allows one to observe transient events in a running chip, in contrast to scan-based analysis, which requires the chip to temporarily suspend functional operation while scanning. The trace logic analyzer supports many functionalities similar to a desktop logic analyzer such as triggering on configurable conditions in real-time, tracing of signals before or after a trigger has occurred, etc. The results of the trace memory can be used for performance monitoring of the processor core or in order to obtain waveforms of internal signals for debugging purposes in the bring-up lab.

The array built-in self-test (ABIST) functionality is used to detect fabrication defects in all on-chip memories (RAM and ROM). The so-called *ABIST engines* apply parallel read and write pattern tests to the memories to detect such faults. If a fault is detected, the ABIST engine has the capability to identify and report the fault, along with information on how to "repair" this fault using redundant bit or word-lines, or other more specific redundancy schemes. The ability to access internal array cells for debugging is also enabled by ABIST.

Logic built-in self-test (LBIST) [9] is used to detect manufacturing faults in the logic. The LBIST controller has a Pseudo-Random Pattern Generator [10] which generates test-patterns. LBIST iteratively uses two different phases: a scan phase which initializes the LBIST scan chain stumps with pseudo-random or random-appearing yet deterministic data, and a functional phase, which clocks the latches for several time-steps. The next LBIST iteration utilizes the scan chain stumps to scan values into multiple-input signature registers (MISRs). In case of a chip with manufacturing faults, the MISR signature will most likely differ from the expected value burned into an unfaulty chip. Verification must ensure that these LBIST phases work as expected; i.e., that all scannable registers can be initialized correctly, that the functional updates do not propagate an uninitialized state to the non-scannable latches, and that the MISR patterns are fully controllable and match in different LBIST modes.

d) Fencing Logic: Fencing logic at chip or partition boundaries ensures that a certain chip or partition is isolated from the surrounding logic while being reset, reconfigured or while running LBIST. The purpose of fencing logic in such a scenario is to provide safe and deterministic input values to the chip / partition. For LBIST, deterministic input values are essential in order to obtain reproducible signatures in the MISRs. Reconfiguration of a partition also requires safe input values in order to guarantee that the final state after reconfiguration is as expected. Fencing logic can be as simple as one logic gate connected to every primary input and a *fence enable* signal that collectively force safe values at internal signals. In other cases, registers are used to implement the fencing logic, which must be initialized to safe values and then forced to hold their state while the fence is enabled.

e) Time Reference: Time reference is an important functionality for systems such as servers with multiple CPUs distributed across different locations. Its purpose is the synchronization of the time-of-day clocks to ensure a consistent time-stamp data across multiple servers and operating systems, e.g., to enable synchronized database accesses.

B. Pervasive Logic Verification Challenges

One challenge of PV is that the design specification is extremely project-specific and the logic is often designed anew for each chip. This results in a specification challenge: the specification cannot be reused to any extent from design generation to generation, unlike many other architectural components of the chip. Another main issue of PV is due to design complexity. In contrast to partition or unit verification, many features of pervasive logic cannot be verified in an isolated block or unit [11]. As the pervasive logic is intertwined with the functional logic, PV has to work with very large components of the design – in cases, even full chip-level models. The cones of influence of the PV properties often span hundreds of thousands of state elements due to aspects such as requiring the use of long serial scan chains [12] and interfacing with very large memory arrays.

The size of the design components required for many PV tasks has historically precluded their receiving any substantial focus from the formal verification community. Furthermore, the sequential depth of many PV tasks – e.g., to serially scan data through possibly hundreds of thousands of latches in a design component – further complicates formal reasoning, e.g., to preclude efficient inductive analysis. Due to these challenges, simulation and hardware emulation have traditionally been used to validate many pervasive features, along with static analysis tools that validate aspects such as scan chain connectivity. While useful for falsification, such approaches are generally incomplete and cannot guarantee the absence of design flaws.

II. Functional Verification Testbenches

The verification paradigm we adopt in this paper is that of a *testbench*, wherein one develops a set of property automata or *checkers* to assess the correct behavior of the design, in addition to a *driver* to constrain the input stimuli to which the design may be subjected to avoid spurious failures. Initialization data is also generally provided for the design. For example, one may initially randomize the state elements of the design, and use the driver to walk the design through a reset sequence prior to performing functional verification. Or, for computational efficiency, one may directly restrict the initial states of the design to those guaranteed by such a power-on reset sequence, without requiring each verification run to begin with performing an explicit reset sequence. The verification task thus consists of trying to obtain a counterexample trace from a specified initial state to one which drives a logical *one* onto the output of a property automata (in the composition of the design with its checkers and drivers), or proving that no such counterexample exists.

Given such a testbench, one may deploy a variety of algorithms to attempt to solve the corresponding properties. For falsification, one may wish to utilize random simulation, hardware emulation, or semi-formal analysis. For verification, one may wish to deploy proof techniques such as rea

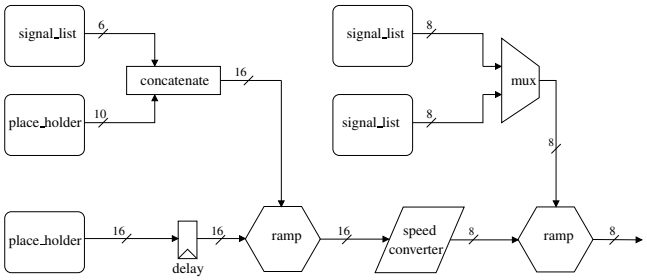


Fig. 1: Example Trace Bus

Primitive Block Type	Description
signal_list	References design signals as data inputs to the trace bus.
mux	Drives the *source* block selected by a user-defined function of a *selector* block.
ramp	OR's two blocks by a user-defined function of an *enable* block.
speed_ converter	This block specifies the transfer of data (another block) across asynchronous clocking boundaries with user-defined behavior.
concatenate	Concatenates other primitive blocks to form a *wider* block.
extract	References only a *subset* of another primitive block.
place_holder	Like **signal_list**; a place-holder for incomplete descriptions.

TABLE I: Primitive blocks used for the *tracedef* language

analysis [13] or induction [14]. To help compensate for the generally exponential resource dependency between the size of the testbench and the resources needed for falsification or verification, various transformation and abstraction techniques have been proposed to automatically reduce this size.

In these experiments, we use the IBM internal verification toolset *SixthSense* [15]. The set of verification and transformation engines we report in our results include the following.

- **COM**: a combinational optimization engine, which attempts to merge functionally equivalent gates and rewrite logic cones to reduce their overall size [16], [17].
- **EQV**: a sequential redundancy removal engine, using a more general set of algorithms to solve a *van Eijk*-style induction problem to identify and merge gates which are sequentially redundant [18], [19], [20].
- **RET**: a min-area retiming engine, which attempts to reduce the number of registers in the netlist by shifting them across combinational gates [21].
- **CUT**: a reparameterization engine, which replaces the fanin-side of a *cut* of the netlist graph with a trace-equivalent, yet simpler, piece of logic [22], [23].
- **LOC**: a localization engine, which isolates a cut of the netlist local to the properties by replacing internal gates by primary inputs [23]. This transformation is sound but incomplete – proofs of correctness on the localized design are valid for the unlocalized design, but counterexamples may be spurious. To help guide the cut-selection process, the engine uses a light-weight SAT-based refinement scheme to include only that logic which is deemed necessary [24].
- **RCH**: a BDD-based reachability engine [25].
- **IND**: a SAT-based induction engine which uses unique-state constraints [26].
- **BMC**: a SAT-based bounded model checking engine [16].

Our system uses a high-performance circuit-based SAT solver with intertwined BDD-based analysis, BDD- and SAT-sweeping for redundancy removal, and structural rewriting algorithms, similar to [16].

III. Trace Bus Verification

The so-called trace and debug bus is a global on-chip bus that enables observation of internal signals for debugging and performance monitoring. This bus is configurable using hundreds of registers, and routes subsets (e.g., 128-bit slices) of many thousands of monitorable points to an on-chip logic analyzer unit. The trace bus represents a major challenge for verification, because the available English specification is often ambiguous, and the straightforward approach to use directed testcases would be extremely time-consuming, lossy in coverage, and error-prone.

We addressed these issues by defining a high-level specification language, called *tracedef*, that allows a simple and concise description of such a bus. This *tracedef* specification is used both for documentation (in place of an English specification) as well as input for automated formal testbench creation. As with the use of any concise formal specification language, the use of this language minimizes the risk of specification errors. Instead of using a more standard language such as PSL [27], we chose to utilize our own language, which provided a more concise description of the trace bus using primitive blocks that closely correspond to the typical logical elements comprising such a bus as described in Table I. These elements also have associated *delays* to reflect those present in the actual design. The tracedef language describes the trace bus as a tree structure. The leaves of the tree are the **signal_list** and the **place_holder** blocks, which model the inputs to the trace bus. The other primitive blocks in Table I represent the tree nodes. Each reference from one block to another results in an edge in the tree, and models a connection in the design. The primitive block representing the root node of the tree represents the output of the bus. Figure 1 illustrates an example structure of such a trace bus.

We developed an automated testbench creation process, which automatically creates drivers and checkers from a *tracedef* specification. The checker is a reference model of the trace bus using library elements for each of the *tracedef* primitives against which the actual trace bus implementation, intertwined with the functional design logic, is checked for equivalence. The driver injects nondeterministic cutpoints at the **signal_list** and **place_holder** elements, though ensures that only valid trace bus configurations are taken into account for this equivalence check. For instance, a multiplexer implementation might use NAND-NOR logic requiring the control register to be one-hot in order to obtain a valid multiplexer behavior, whereas other implementations may support arbitrary selector values. Our testbench creation process extracts all valid selector values from the *tracedef* specification, and constrains the testbench driver to allow only those valid settings. Additional configuration data, which we term *traceconf*, are used to specify constraints for primary inputs such as clock frequencies, necessary reset signal setup, constant enable signals, etc.

The design we verified posed an additional challen

Metric	Initial	COM	LOC[1]	COM	EQV	Resources
Inputs	33441	21492	11	11	0	792s 624MB
Gates	924723	797710	596	493	0	
Registers	142072	125520	193	193	0	
Properties	128	128	1[1]	1	0	

TABLE II: Memory Flow Controller Results. [1]Localization performs a case split, solving each property independently; the largest localized cone is reported. A column with 0 properties reflects that the corresponding engine solved the problem. Resources reported are cumulative for all 128 properties.

Metric	Initial	COM	LOC[1]	CUT	RET	COM	Resources
Inputs	4188	2878	173	151	238	0	14508s 412MB
Gates	271270	144559	857	1193	1292	0	
Registers	83880	33322	370	370	109	0	
Properties	1381	700	1[1]	1	1	0	

TABLE III: Load Store Unit Results. [1]Localization performs a case split; the largest localized cone is reported. Resources reported are cumulative for all properties.

it uses multiple latch and register types, which are sensitive either to the rising or the falling edge, or various levels, of the clock. This complicated the equivalence check using our simple automatically-generated reference model, which consisted of just one register type for simplicity. We solved this problem by constraining the driver cutpoints at the trace bus data inputs to arbitrary nondeterministic values that cannot toggle more frequently than once per clock period. This is a conservative constraint since in the actual design, these signals are each driven by one type of state element hence cannot toggle more frequently than once per clock period. With this constraint, we eliminated spurious mismatches due to our reference model sampling the cutpoints at a different clock phase than the actual design.

A. Verification Results

One of the larger design slices that we specified and tested using the described methodology is that of a Memory Flow Controller subsystem. This subsystem includes an L2 cache which is inclusive of L1 cache data, and non-cacheable units, which handle cache-inhibited requests from the processor core. There is a controller associated with the cache which handles various operations including the cacheable load and store requests from the core and the memory management unit.

For the memory flow controller there are 128 properties, which are one-bit reference-model checks corresponding to each output of the 128-bit trace bus. Because the trace buses do not route data from or through memory arrays, to simplify the formal verification task we black-boxed all arrays, replacing their output ports by nondeterministic cutpoints. Table II provides the results of solving these properties with a multi-algorithm flow. We report problem size in terms of the number of nondeterministic input variables, combinational gates (in terms of synthesis down to 2-input AND gates), registers, and properties. In addition to a relatively large netlist size, the memory flow controller spanned nearly 16,000 lines of design VHDL. If all of the library files used by the memory flow controller are included, there are more than 300,000 lines of VHDL. As discussed in Section I, the trace buses were intertwined with the design logic in the VHDL, complicating the overall verification task.

Nonetheless, as illustrated by Table II, the ability to leverage the proper algorithm flow enabled us to solve these problems efficiently in approximately 13 minutes. All experiments reported in this paper were run using a single processor of a 16-way 1.9GHz POWER4 system. The optimal solution first employed low-cost combinational optimization across all properties, simplifying subsequent localization analysis. After a dramatic reduction through localization, each subsequent localized property was solved efficiently using combinational optimization followed by van Eijk-style induction. Though each property represented a bit-slice of the trace bus, isomorphisms among the properties were broken at numerous points due to intertwined BIST chains and lack of isomorphisms among the functional logic being sampled by the trace bus.

We also applied this methodology in numerous places in the processor core. For example, we applied the technique to the load-store unit, again black-boxing the larger memory arrays for reduced resources. The results of this verification are provided in Table III. Though somewhat smaller than the memory flow controller, this run took nearly 4 hours to complete. The longer run-times were partially attributed to the larger number of properties, and partially to more complex control logic requiring more post-localization transformations before proofs became feasible. These transformations included reparameterization and min-area retiming. After retiming, the resulting problem became a tautology easily discharged by combinational optimization. In contrast, without localization, retiming was unable to sufficiently simplify the problem to render tautologies. Without these transformations, induction alone was very expensive and could not solve the properties within 48 hours.

Dozens of design flaws were encountered during these efforts. The most basic, as would be expected, were that incorrect signals were propagated through the debug bus with incorrect timing as compared to the specification, typically due to improper multiplexor selector implementations or bad wiring of the bus. Some of the more intricate flaws were due to clock gating controls disabling certain latches when they were needed to route data, or speed conversion logic sampling signals with improper timing.

IV. ABIST Verification

Array Built-In Self-Test (ABIST) logic is used in a chip to identify manufacturing defects such as stuck-at faults or short circuits in a memory array [28]. Such logic consists of an *ABIST engine* connected to one or more arrays. The ABIST engine drives address and control information, along with specific write-data patterns, into the scan latches adjacent to the array. The chosen write-data patterns are carefully selected so as to attempt to cover all possible fault types as efficiently as possible. The ABIST engine then triggers the writing of the scanned data into the array using dedicated communication latches. These latches act as pipeline stages to enable the shared ABIST engine to reside further from the arr

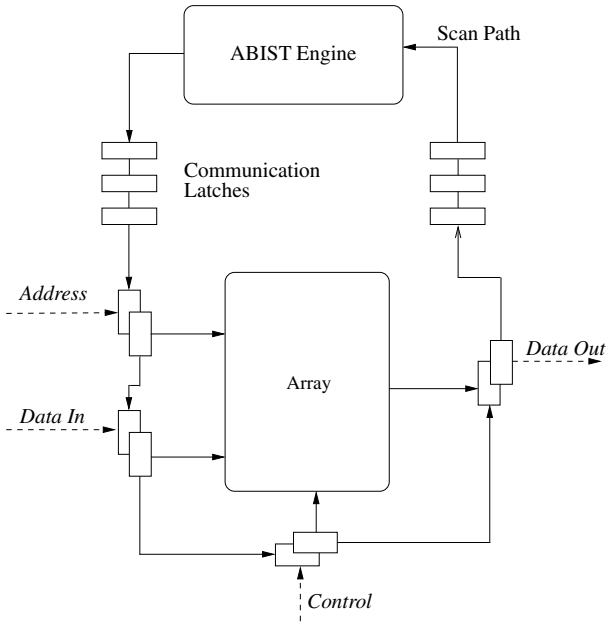


Fig. 2: ABIST Design

than otherwise possible in silicon due to timing requirements and circuit delay.

After sequentially writing all rows and columns of the array, the ABIST engine next reads the written content out of the array in the same order that it was written. This read-out data is then compared against the expected data. Any deviation from the expected data is reported as a fault in the array structure.

Using such a scanned ABIST approach, the ABIST engine cannot test the array faster than the number of clock periods necessary to serially scan all address, control and data bits required to trigger the writes and reads of all necessary patterns and all array cells. In order to alleviate this problem, *shadow* scan latches are added for the address and control path of the array. These latches are interleaved to allow multiple addresses to be applied on successive clock cycles, eliminating the bottleneck of the serial communication channel between the ABIST engine and array. While merely an optimization technique for the ABIST performance, such constructs significantly increase the complexity of the ABIST logic and its verification.

Many modern array implementations provide *repair* capability: if a *repairable* fault is detected, the array may utilize redundant bit or word lines to mask a specific number of repairable faults for subsequent chip operation [29].

Verification of ABIST logic typically consists of artificially seeding errors into the array, and validating that the ABIST engine correctly identifies them, and if applicable, provides information on how to repair them. The main goal of leveraging formal analysis in ABIST verification was to be able to exhaustively test the response of the ABIST engine against the very large set of possible error types. To speed up verification, many ABIST implementations have the ability to program the ABIST engine to cover only specific portions of the arrays: e.g., a smaller set of rows and columns than the large number of rows and columns that comprise a cache. We exploited this capability when performing verification of the ABIST design. Nonetheless, due to the sheer size and sequential depth of these verification runs, attempting any form of exhaustive simulation becomes computationally infeasible. Though these factors also posed great challenges to formal analysis, the availability of the proper set of algorithms ultimately enabled symbolic algorithms to scale to this task.

A. Verification Results

To compensate for the sheer size of the ABIST logic, comprising an entire L2 cache plus the dedicated scan and controller latches, in addition to the ABIST engine itself, our formal testbenches programmed the ABIST engine to operate only upon specific slices of the L2 cache. Without operating on single slices alone, the size of the given testbench was 5,321,918 state elements and 32,143,002 gates. The smallest testbench we could obtain for an array slice, which was still big enough to provide meaningful verification results, contained 246,302 state elements.

The properties we developed for the testbench checked that:

- The state machines of the ABIST engine properly transition from write phase to read phase to compare phase, and finally properly report the end of the testing phase.
- The ABIST engine properly "times" the sending of data to the individual arrays given the pipeline depth of the communication channels between them.
- The ABIST engine properly communicates with the proper arrays.
- Injected errors are properly detected by the ABIST engine.
- The ABIST engine properly indicates whether a repairable number of errors was detected.

Because the goal of the ABIST engine is to properly detect errors, our testbench performed single- and multiple-bit error injection by randomly selecting array cells to "corrupt" by altering the data written by the ABIST engine in transit to the arrays prior to the read and compare phases.

Table IV summarizes our verification results for one slice. Due to the nature of this logic, few transformations were useful to reduce it; isomorphisms were broken by the nature of the ABIST engine and pipeline stages, and there was little redundancy to be exploited. More aggressive transformations such as sequential redundancy removal were time-consuming given the size of the logic, and not very powerful in their reductions. Reachability was clearly infeasible in this domain, and due to the sequential depth of the ABIST engine's sequence, induction also became infeasible. We thus resorted to *bounded* unreachability analysis using BMC. Luckily, since the duration of the ABIST engine process is readily quantifiable, this bounded unreachability approach provided full confidence of correctness of the checked properties.

The behavior of the ABIST engine is largely deterministic; once triggered, it walks through a long execution stream. The primary sources of nondeterminism include the black-boxing done to prune the testbench down to a single slice, and the random selections for error injection. The six propert

Metric	Initial	COM	BMC 440,000	Resources
Inputs	29176	29086	0	
Gates	1567812	1394640	0	1611s
Registers	246302	230495	0	5976MB
Properties	6	6	0	

TABLE IV: ABIST Engine Results

Metric	Initial	COM	EQV	IND	Resources
Inputs	548878	32362	843	0	
Gates	748426	245309	43978	0	211s
Registers	73368	23560	5922	0	748MB
Properties	4665	4665	1837	0	

TABLE V: Fencing Logic Results

Table IV include validation that the ABIST engine cycles through its write, read, and compare phases properly, and that errors are properly identified by the ABIST engine. Note that we completed a bounded analysis of 440,000 time-steps for these checks. The "ABIST check done" occurred at time 30,407, giving us confidence that the ABIST engine properly detected all forms of failures. For this reason, we could have concluded that a 30,407-step BMC was adequate to infer correctness - though analysis beyond the critical time-frames was trivial given the nature of our BMC engine.

Our BMC engine uses a structural SAT solver applied in an incremental manner, time-step by time-step. It is highly tuned to unfold only critical signals at critical time-steps, leveraging the simplification performed at earlier time-steps to reduce the size of unfoldings of later time-steps in addition to reusing learned clauses. For ABIST verification, the actual symbolic reasoning about the unfolded instance is not very difficult for the SAT solver; the key was tuning the BMC infrastructure for large designs and very deep unfoldings. For time-frames beyond that which the ABIST engine was operating, structural analysis alone detected that the unfoldings were trivial. It is worth noting that, after tuning our SAT solver in this manner, BMC became several orders of magnitude faster than even random simulation for this large design, since the latter could not be as optimally tuned to operate only on critical portions of the design over time.

Numerous design flaws were exposed during the ABIST testing, including faulty reporting of array errors and incorrect staging of communication channels between the ABIST engine and arrays.

V. Fencing Logic Verification

Fencing logic is typically a small yet essential part of systems with multiple chips or multiple clock domains. The intent of such logic is to prevent spurious logic activity while the system is in the process of being reset, reconfigured or while running LBIST. For example, when an incoming fence signal is active for a particular domain, the internal logic of the receiving domain must be impervious to random transitions that may occur on any number of incoming interface buses or signals [30].

Verification of fencing logic requires demonstrating that all logic associated with a particular fence or set of fences is effectively quiesced during any window of time that the fences are active. For optimality of the design, the fencing logic itself is often kept minimal, risking the exposure that certain input stimulus may erroneously sensitize transitions in the fenced design. Since a single fence line typically serves to protect a multitude of interface signals from interacting with a large amount of downstream logic, simulation alone is often insufficient to expose all possible logic interactions and flaws. This renders simulation-based approaches largely insufficient to yield acceptable coverage, and motivates a formal verification approach.

Due to the straight-forward nature of the verification task, we were able to develop a largely automated formal testbench creation paradigm as follows.

- A property is automatically generated for each register in the fenced design region, checking that the register's value does not alter during the fencing condition.
- All fencing-control inputs are configured to enable fencing.
- The remaining design inputs are categorized from the design specification as being *fenceable* vs. *unfenceable*. Fenceable inputs are precisely those which the fencing logic is required to shelter the design from being sensitized to, hence these inputs are left unconstrained in the testbench. Others may come from adjacent logic blocks which themselves are to be fenced, hence these are driven to arbitrary constant values to avoid rendering spurious failures.

A. Verification Results

We deployed our fencing logic verification methodology on a custom data-flow chip. For our first deployment, before we had a well-tuned sequential redundancy removal engine, we limited our verification to 21 individual units of that chip, each of which was limited to several thousand state elements which were solvable within 10 minutes. Unit-level inputs from other fenced blocks were treated as *fenceable* as per the above methodology, and driven constant; all others were randomized.

Later in the project, a highly-tuned **EQV** engine scalable to larger designs became available. This engine trivialized those unit-level runs to being solvable within a matter of seconds. We thus found that we were able to apply this methodology at the level of the entire chip, though black-boxing the logic arrays since that had no impact on the fenced logic. With this methodology, we were able to leave all chip inputs nondeterministic (aside from those enabling fencing), relying upon the inter-unit connections to effectively propagate the fenced constants throughout the chip. This saved manual effort, since we no longer had to categorize inputs as fenceable vs. non-fenceable. This also eliminated the risk of missed design flaws associated with mis-categorization. Table V provides the results of these experiments.

Six design flaws were identified during this testing, wherein the fencing logic was inadequate to prevent the sampling of design input values into the fenced logic.

VI. External Time Reference (ETR) Verification

ETR denotes a mechanism to keep all processor cores in all nodes of a system synchronized to the same

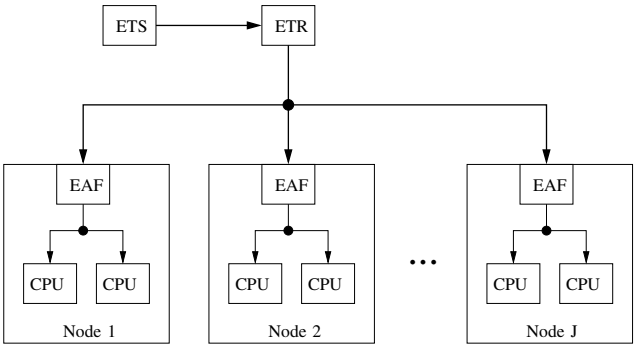


Fig. 3: ETR System

"Time Of Day" [31]. Such a mechanism is very important, for example, in synchronizing database accesses. Each node in a system is connected to the ETR via the ETR Attachment Facility (EAF). The ETR itself is connected to an External Time Source (ETS), usually an atomic clock. Such a system is illustrated in Figure 3.

The ETR generates timing information from the ETS, encodes it and sends it serially in the form of a bit stream to each EAF. For enhanced reliability, this information is redundantly transmitted across two identical channels, and the ETR can dynamically switch between channels if it detects errors. The EAF decodes this bit stream from the selected channel, stores the results and propagates them to the individual CPUs.

Each EAF channel has two main components: a decoder and a store unit. In the decoder, the incoming stream is first sampled at the frequency of the local clock, then passed to an edge-detection unit to extract a bit stream from the samples. The extracted bits are then packed into 16-bit symbols by a deserializer. Afterwards, the decoded symbols are passed to the store unit, which categorizes them as control vs. data words of the following types. The data symbols include Data Byte Symbols (DBS) and Longitudinal Redundancy Check symbols (LRC), which serve as a parity-check for the DBS. The control symbols include the following:

- Idle (IDL), which is used to synchronize the EAF to the incoming stream.
- Data frame start (FST), which is used to flag the start of a data frame containing DBS symbols.
- On-time symbol (OTS), which is used to indicate the end of the stream, and triggers the propagation of the timing information decoded by the EAF channel.

The ETR periodically sends a pattern composed of these symbols, beginning with a sequence of IDL symbols and terminated by one OTS. Each pattern may contain numerous duplicates of the data frame. The rest of the pattern is filled with IDL symbols. Figure 4 illustrates such a pattern.

Before the channel stores and begins processing the timing data, it needs to be synchronized to the incoming ETR stream. This synchronization is achieved by detecting and aligning against a long sequence of IDL symbols. Once synchronization is achieved, the data symbols are stored and validated by comparing the received LRC symbol to a locally-generated value. If the comparison matches, the data is considered to be valid and the subsequent received data frames are ignored,

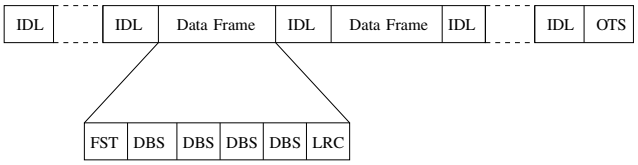


Fig. 4: Example ETR Pattern

since they are only duplicates. If the comparison fails, the process is repeated with the duplicate data frames until valid data is detected. On detection of an OTS symbol, successfully stored data is propagated to the processors. Otherwise, the cause of the failure is reported, such as failure to synchronize, data errors, or missing OTS.

A. Verification Results

The type of properties we verified of the ETR included aspects such as correct symbol decoding, correct synchronization to the incoming signal, proper storage and propagation of data, and proper error detection and handling.

A black-box approach of verifying the overall ETR and EAF logic was quite challenging for formal verification. This system comprised a large number of state elements even after all transformations had been deployed, and was sequentially very deep. A white-box approach was therefore adopted using two testbenches: the first testbench deals with the serial part of the unit, namely the decoding of the bit stream. This testbench is used to validate properties for symbol decoding and synchronisation. The second testbench deals with the parallel part of the unit, namely the storing of the timing information and the switching between the two channels. This testbench is used to validate properties about proper data storage and error handling. The actual design logic was identical across both testbenches; however, the drivers and checkers tied in to the design at different points.

One example property verified of the serial testbench is that every symbol coming from the ETR is correctly decoded. The driver of our testbench was configured to drive a stream of random symbols to the ETR. Because the stream is entirely nondeterministic, we constrained our testbench to validate only the decoding of a single arbitrary symbol. Because the necessary amount of time to decode a symbol was fixed, a BMC approach was adequate to complete this verification task. Nonetheless, the depth of the check was quite large, primarily because the clock frequencies across the ETR and EAF differ by a factor of 96 (hence we used oscillators of periodicity 2 and 2×96 in our driver to clock the two components). The results are depicted in Table VI.

One example property we verified of the parallel testbench is that the detection of the OTS occurs properly. In particular, we built a driver that randomly determines the length of streams it will send from the first data frame to the final OTS (refer to Figure 4). The property verified that if the OTS occurs outside of the allowed tolerance, a missing OTS interrupt is generated. The results are depicted in Table VII. Because the ETR system is architected to be extremely robust, the timeout period for a missing OTS is quite high, he

Metric	Initial	COM	EQV	COM	BMC 6,162	Resources
Inputs	2957	18	16	16	0	
Gates	72340	49489	3460	3264	0	843s
Registers	9544	5576	725	725	0	
Properties	3	3	3	3	0	324MB

TABLE VI: ETR Serial Property Results

Metric	Initial	COM	EQV	RET	COM	RCH	Resources
Inputs	319	296	26	38426	26	0	
Gates	73523	49727	798	87196	778	0	78028s
Registers	12922	8720	3332	1725	1724	0	
Properties	10	10	10	10	10	0	8.4GB

TABLE VII: ETR Parallel Property Results

reachability computation required 17,698 image computations for convergence.

VII. Concluding Remarks

Pervasive logic refers to a variety of functionality included in hardware chips orthogonal to their primary architectural functionality, such as logic for initializing the design, for security purposes, and for self-test. Verification of pervasive logic is challenging for a variety of reasons, including the facts that pervasive logic is entirely customized for each chip hence verification and specification reuse tend to be infeasible; the pervasive logic is intertwined with the functional logic of the design at all hierarchies; and most pervasive logic verification tasks require very large design slices (10,000s to millions of state elements) and require very deep temporal analysis. These complexities have historically limited pervasive logic verification to being amenable only to simulation or emulation-based analysis.

In this paper, we discuss how a robust multi-algorithmic formal framework has made a variety of pervasive logic verification tasks feasible and efficient. Given the magnitude in terms of size and sequential depth of many of these tasks, without the right set of algorithms, formal analysis would be infeasible. We in many cases needed to tune our algorithms to better scale to the necessary magnitudes for these applications. These formal solutions have made a substantial improvement to methodologies for verifying such pervasive logic for present and future designs.

While we have made significant strides in this direction, we note that pervasive logic verification is still far from a solved problem. Numerous pervasive verification tasks remain far outside the realm of proof capability, due to requiring the analysis of extremely large design components – possibly entire processor cores and even chips – for which sufficiently advanced algorithms are not available. Though traditionally addressed in distinct *test* conferences and conference tracks, we thus wish to introduce the challenges of pervasive logic verification to the formal verification community. In particular, we wish to encourage increased attention to applicable methodologies and continued research in the development of larger-capacity automated proof algorithms to formally address such pervasive-logic verification needs.

References

[1] D. Pham, "Key features of the design methodology enabling a multi-core SoC implementation of a first-generation Cell Processor," in *Asia and South Pacific Design Automation Conference*, Jan. 2006.

[2] S. Thakkar, "Second-generation Intel Centrino mobile technology," in *Intel Technology Journal*, vol. 9, Feb. 2005.

[3] J. Ludden et al., "Functional verification of the POWER4 microprocessor and POWER4 multiprocessor systems," *IBM Journal of Research and Development*, Jan. 2002.

[4] S. Chin, "IBM's eFuse technology portends adaptable chips," in *EE Times*, July 2004. http://www.eetimes.com/showArticle.jhtml?articleID=26100962.

[5] A. Orlowski, "The Cell chip - what it is, and why you should care," *The Register*, Feb. 2005. http://www.theregister.co.uk/2005/02/01/cell_analysis_part_one.

[6] IEEE Standards Board, *IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture*.

[7] M. Schwerdtfeger, "SPI - serial peripheral interface," June 2000. http://www.mct.net/faq/spi.html.

[8] Philips Semiconductors, *The I^2C-Bus Specification Version 2.1*. Jan. 2000. http://www.semiconductors.philips.com/acrobat_download/literature/9398/39340011.pdf.

[9] G. A. Van Huben, "The role of two-cycle simulation in the s/390 verification process," *IBM Journal of Research and Development*, vol. 41, no. 4/5, 1997.

[10] D. Das and N. A. Touba, "Reducing test data volume using external/LBIST hybrid test patterns," in *International Test Conference*, 2000.

[11] C. Stroud and G. Liang, "Design verification techniques for system level testing using ASIC level BIST implementations," in *IEEE International ASIC Conference and Exhibit*, 1993.

[12] D. Chang, M.-C. Lee, K.-T. Cheng, and M. Marek-Sadowska, "Functional scan chain testing," in *DATE*, Feb. 1998.

[13] O. Coudert, C. Berthet, and J. C. Madre, "Verification of synchronous sequential machines based on symbolic execution," in *Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[14] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *FMCAD*, Nov. 2000.

[15] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, Nov. 2004.

[16] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Transactions on CAD*, Dec. 2002.

[17] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *DAC*, July 2006.

[18] C. A. J. van Eijk, "Sequential equivalence checking without state space traversal," in *DATE*, March 1998.

[19] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *FMCAD*, November 2000.

[20] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *DAC*, June 2005.

[21] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *CAV*, July 2001.

[22] I.-H. Moon, H. H. Kwak, J. Kukula, T. Shiple, and C. Pixley, "Simplifying circuits for formal verification using parametric representation," in *FMCAD*, Nov. 2002.

[23] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies," in *CHARME*, Oct. 2005.

[24] D. Wang, *SAT based Abstraction Refinement for Hardware Verification*. PhD thesis, Carnegie Mellon University, May 2003.

[25] I.-H. Moon, G. Hachtel, and F. Somenzi, "Border-block triangular form and conjunction schedule in image computation," in *FMCAD*, Nov. 2000.

[26] N. Een and N. Sörensson, "Temporal induction by incremental sat solving," in *BMC*, 2003.

[27] Accelera, *PSL LRM*. http://www.eda.org/vfv.

[28] C. T. Mo, C. L. Lee, and W. C. Wu, "A self-diagnostic BIST memory design scheme," in *Memory Technology, Design and Testing*, Aug. 1994.

[29] V. Schober, S. Paul, and O. Picot, "Memory built-in self-repair using redundant words," in *Int'l Test Conference*, Nov. 2001.

[30] E. Hallee, S.-L. Huang, K. Hwang, and B. Messina, "Fencing circuit and method for isolating spurious noise at system interface," in *U.S. Patent 5,386,393*, 1993.

[31] F. Injey, "External time reference (ETR) requirements on z990," in *IBM Redbooks Flash REDP-3753-01*, Feb. 2004.

Post-reboot Equivalence and Compositional Verification of Hardware

Zurab Khasidashvili, Marcelo Skaba, Daher Kaiss, Ziyad Hanna
Intel, IDC, Haifa, Israel
{zurabk, smarcelo, dkaiss, zhanna}@iil.intel.com

Abstract

We introduce a finer concept of a Hardware Machine*, where the set of post-reboot operation states is explicitly a part of the FSM definition. We formalize an ad-hoc flow of combinational equivalence verification of hardware, the way it was performed over the years in the industry. We define a concept of* post-reboot bisimulation*, which better suits the Hardware Machines, and show that a right form of combinational equivalence is in fact a form of post-reboot bisimulation. Further, we show that alignability equivalence is a form of post-reboot bisimulation, too, and the latter is a refinement of alignability in the context of compositional hardware verification. We find that post-reboot bisimulation has important advantages over alignability also in the wider context of formal hardware verification, where equivalence verification is combined with* formal property verification *and with* validation of a reboot sequence*. As a result, we propose a more comprehensive, compositional, and fully-formal framework for hardware verification. Our results are extendible to other forms of labeled transition systems and adaptable to other forms of bisimulation used to model and verify complex hardware and software systems.*

1. Introduction

This work addresses formal hardware verification. The aim of hardware *equivalence verification* is to check for "functional equivalence" of two design models according to some concept of equivalence. The equivalence of two models does not guarantee that they do what they are designed for, and it is a task of formal *property verification* and dynamic validation to provide a level of confidence that the designs have the desired functionality. In practice, equivalence verification and property verification are closely related and are often performed under a common methodology umbrella and tool set. To date, however, there has not been any significant research towards providing a unifying theory that would combine equivalence verification with property verification using *fully formal, full-proof methods* (and not relying on a non-exhausting simulation).

Since verification of complex hardware is not imaginable without employing abstraction and compositional methods, hardware equivalence verification in practice consists in the following steps:

1. Decompose the specification and implementation models using mapped cut points in them.
2. Use boundary constraints to make the corresponding component slices equivalent.
3. Build a reboot sequence that via 3-valued simulation [HC98] brings the models into states satisfying the boundary constraints (and possibly other properties).
4. Check that all properties remain valid post-reboot, using (non-exhaustive, 3-valued) simulation.

The most widespread equivalence verification method in the industry was and still is some form of *combinational verification* [KvE04], where the slices are combinational, i.e., they contain no internal state elements. Hardware is represented as a Finite State Machine (FSM) [Koh78, HS96]. In its classical definition, combinationally equivalent FSMs M_1 and M_2 are actually the same FSMs: their transition relations and output functions are defined via the same Boolean functions that may be implemented differently in M_1 and M_2. In practice, however, a form of assume-guarantee framework is used (like the one outlined above), where the requirement of component equivalence is weakened to a form of conditional equivalence under "don't cares", or under combinational or temporal logic assumptions. Thus, combinational equivalence in practice does not correspond to its classical definition.

For hardware FSMs designed to operate correctly after simulating them with a *reboot sequence*, several concepts of equivalence have been developed (besides combinational equivalence), such as *sequential hardware equivalence,* also called *alignability equivalence* [Pix92], *delayed safe replaceability* [SPAB01], *exact 3-valued equivalence* [RSSB99], and *steady-state equivalence* [KH02]. The latter two forms of equivalence have been used for *retiming verification* [LS91], which is the second most widespread form of hardware equivalence verification. Retiming verification of sequential components is often combined with combinational verification of combinational components. It is unclear what kind of equivalence is proved between the specification and implementation FSMs as a result of such a combination of verification methods. Furthermore, step 4 of the above outlined procedure of compositional verification, which relates reboot sequence with the used boundary properties, was never considered as part of formal equivalence verification. Indeed, it is currently based on a non-exhaustive dynamic simulation and thus cannot guarantee a full proof.

The main results of this work are:

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

1. *From practice to theory*: We formalize several dominating hardware equivalence verification methods into a unified theory (we identify and fill the verification holes along the way) – the outcome is *post-reboot equivalence.*
2. *From theory to practice*: We propose (fully) formal, practically applicable hardware equivalence verification algorithms and methodology allowing combination of formal equivalence verification with formal property verification.
3. *Relevance in practice*: We present experimental evidence demonstrating that the new theory makes difference in the practice of full-chip verification.
4. *Extendible and adaptive concepts*: While we focus on *bit-wise* hardware machines, the new equivalence concept is defined in terms of bisimulation, which allows (1) extending the results to (possibly non-deterministic) Labeled Transition Systems, into which both hardware and software can be modeled at higher levels of abstraction, and (2) adapting it to other useful forms of (bi)simulation.

The concept of post-reboot equivalence proposed here is a refinement of alignability equivalence [Pix92]. The compositional verification framework that we propose is based on a more recent work [KSKH04] that proves *weak compositionality* of alignability via defining a concept of *stable decomposition* of the specification and implementation FSMs. The latter framework assumes, however, that the two FSMs are weakly synchronizable (WS for short [PR96]) with the same input sequence, but it does not provide any practical algorithm for formally verifying whether or not a given input vector sequence is a ws-sequence. This makes the methodology proposed in [KSKH04] incomplete. This incompleteness is caused by the fact that the latter work used the alignability equivalence as the basis, and it is unclear how a practical, formal verification method can be proposed for verifying ws-sequences without adopting the post-reboot equivalence concept, as we do here. Furthermore, alignability equivalence is not satisfactory in practice, since it is not enough for a reboot sequence to be a ws-sequence: to make the hardware work properly, the reboot sequence should bring it to a designated set of states that meet some architectural requirements. For example, physical addresses in the memory, initial values of counters, and some states at the internal sub-systems must have specific values. The concept of WS does not capture these requirements. Finally, alignability equivalence is not expressive enough to relate the provability of commonly observable temporal logic formulas in one FSM to their validity in an equivalent FSM. These points will later be clarified in detail.

The theoretical contribution of our work can briefly be summarized as follows: We introduce a concept of *post-reboot bisimulation* as a pair (π, B), where B is a bisimulation between compatible FSMs M_1 and M_2 (i.e. the FSMs have the same inputs and outputs) and π brings any pair of states of $M_1 \times M_2$ into a pair in B. We formalize the concept of *post-reboot combinational equivalence* and show that it is a post-reboot bisimulation. We show that alignability is a post-reboot bisimulation, too. We also show that the set of all post-reboot bisimulations (when it is non-empty) forms a complete lattice [DP90]. Its top element corresponds to bisimulations formed from all ws-states, and the bottom element corresponds to the bisimulations formed from the states in *sink strongly connected components* [PR96] of M_1 and M_2. Post-reboot combinational verification and compositional alignability verification correspond to building post-reboot bisimulations that are between the top and bottom bisimulations, and therefore these forms of verification are feasible in practice.

Further, we define an upper semi-lattice [DP90] of weak-synchronizing sequences. It is in fact this order that allows us to demonstrate how "the strength" of a reboot sequence can affect the post-reboot validity of a temporal specification of the design – and this indeed clarifies the subtle differences between post-reboot equivalence and alignability: Since a reboot sequence π must bring any state pair of $M_1 \times M_2$ into a non-empty bisimulation B, the sequence π must be a ws-sequence for both M_1 and M_2. The converse need not be true: some of the ws-sequences cannot serve as useful reboot sequences because they may not meet some *non-functional* requirements that should be satisfied during the post-reboot operation of the circuit. Examples of non-functional requirements (for the FSM formalism) are power and timing constraints, as well as the architectural requirements mentioned above, which may be satisfied in some but not all ws-states. Non-functional requirements can also be temporal logic specifications that were not encoded into the circuit design as observable output behaviors. Thus, post-reboot bisimulation can be viewed as a useful *refinement* of alignability equivalence, especially when equivalence verification is considered in a wider context of formal verification of hardware designs, and may be combined with verification of temporal properties and validation of (candidate) reboot sequences.

In the next section, we recall the FSMs and concepts related to alignability and introduce Hardware Machines. In section 3, we introduce post-reboot bisimulation, relate it to alignability, and give its lattice-theoretic characterization. In Section 4, we propose a revised definition of combinational equivalence. We discuss the advantages of post-reboot bisimulation for verification of hardware machines in Section 5. Our conclusions appear in Section 6.

2. Hardware Machines

In this section, we introduce Hardware Machines, to reflect the fact that the set of operation states of a hardware design is a subset, usually proper, of the WS-states.

Definition 2.1 [Koh78] A Finite State Machine (FSM) M is a tuple $(S,\Sigma,\Gamma,\delta,\lambda)$, where S is a finite set of states (ranged

over by s, t, s_1, …); Σ is a finite input alphabet (ranged over by a, …); Γ is a finite output alphabet (ranged over by e, …); $\delta: S \times \Sigma \rightarrow S$ is a state transition (or next-state) function; and $\lambda: S \times \Sigma \rightarrow \Gamma$ is an output function. Here, Σ and Γ correspond to the Boolean vectors of input and output variables (i.e., bits that can be 0 or 1) and, similarly, states are Boolean vectors of state (i.e., latch) variables.

Notation: Below, unless otherwise stated, M_1 and M_2 denote *compatible* FSMs, i.e., M_1 and M_2 have the same sets of inputs and outputs. We denote the state set of M by S(M). Further, π and ρ denote input sequences for M. We write $a: s \rightarrow t$ if $\delta(s,a) = t$, and we write $\pi: s \rightarrow^* t$ if π transforms s into t (here $\rightarrow^*$ denotes transitive reflexive closure of →). Recall that $a: s \rightarrow t$ means that input a brings M from state s (the current state) to state t (the next state); and $\lambda(s,a) = e$ means that at current state s, if the input is a, the output value (at current state) is e. Finally, $O_M(s,\pi)$, or simply $O(s,\pi)$, denotes the output value of M after simulating M with π, where M is initially at state s, and $S(s,\pi)$ denotes the state into which π brings s.

We recall that the *product* $M_1 \times M_2$ of M_1 and M_2 has the same inputs and outputs as the two FSMs; its states are pairs of states of M_1 and M_2, and its output function and state transition function are pairs of the output functions and state transition functions of M_1 and M_2, respectively.

Definition 2.2

- [Koh78, HS96] Let M_1 and M_2 be FSMs. States $s_1 \in S(M_1)$ and $s_2 \in S(M_2)$ are *equivalent*, written as $s_1 \approx s_2$, if $\forall \pi: O(s_1, \pi) = O(s_2, \pi)$. The state (s_1,s_2) is then called an *equivalent state* of the product machine $M_1 \times M_2$.
- [PR96] A *weak synchronizing sequence* (*ws-sequence* for short) of an FSM M is an input sequence that brings M from any state to a subset of equivalent states $\{s_1,\ldots,s_m\}$. Each such state s_i is called a *ws-state* of M, and M is called weakly-synchronizable.

Below, $StateEq(M_1,M_2)$, or simply StateEq, denotes the state equivalence relation on $S(M_1) \times (M_2)$; ≈ is the infix notation for StateEq. By equivalent states we always mean state-equivalence. Note that every state reachable from a ws-state of M is a ws-state. WS(M), or simply WS, denotes the set of all ws-states of M.

Definition 2.3 [Pix92] Let M_1 and M_2 be FSMs.

- A binary input sequence π is an *aligning sequence* for a state (s_1,s_2) of $M_1 \times M_2$ if it brings $M_1 \times M_2$ from state (s_1,s_2) to an equivalent state.
- M_1 and M_2 are *alignable*, $M_1 \approx_{aln} M_2$, if every state of $M_1 \times M_2$ has an aligning sequence.

It is shown in [Pix92] that $M_1 \approx_{aln} M_2$ iff there is a sequence (called a *universal aligning sequence)* that aligns each state of $M_1 \times M_2$. The following theorem is an easy consequence of the results of [Pix92].

Alignment Theorem: FSMs M_1 and M_2 are alignable if and only if each FSM is weakly synchronizable and there is an equivalent pair $s_1 \approx s_2$ of states in M_1 and M_2. The concatenation of ws-sequences of M_1 and M_2 is a ws-sequence for both of them and it weakly synchronizes M_1 and M_2 into equivalent ws-states (when $M_1 \approx_{aln} M_2$).

If an FSM is not weakly-synchronizable, for any input sequence ρ, there always exist power-up states s_1 and s_2 such that the states $O(s_1, \rho)$ and $O(s_2, \rho)$ are not equivalent. This means that, whatever the ρ, the FSM exhibits a non-deterministic observational (output-) behavior after ρ. Therefore, in the alignability equivalence, weak synchronization is a necessary condition for an FSM to be equivalent to another FSM (or to itself). Below in the discussion, we will only consider such FSMs.

Since alignability equivalence is only concerned with the output behavior, and equivalent states of an FSM cannot be distinguished by observing the outputs, Pixley [Pix92] worked with equivalence classes of states $[s]_{\approx}$. He showed that, for any non-equivalent ws-states s and t, there is a transition path from (an element of) $[s]_{\approx}$ to (an element of) $[t]_{\approx}$ and vice versa. Hence, whatever the ws-sequence π is chosen to synchronize an FSM M, the set of its post-ws states, up-to ≈, is always the same – it coincides with the set of equivalence classes of WS(M). Thus, for alignability equivalence, the set of all ws-states is (implicitly) considered as the set of operation states.

In practice, equivalence verification between FSMs M_1 and M_2 is usually combined with verifying that the specification model M_1 (written in a hardware description language) satisfies its temporal logic specification, say P. In this wider context, working with equivalence classes of states is inadequate, as if a state s satisfies P, its equivalent states need not to. And for ws-states, it is no longer valid that there is a transition path between any pair of ws-states. Therefore, it does make a difference which ws-sequence is chosen to weakly synchronize the FSMs – the resulting sets of post-ws operation states may be different. For one ws-sequence π, the respective post-ws operation states of M_1 may satisfy P while for another ws-sequence ρ, the resulting post-ws operation states might not satisfy P. A simple example of this is given in Section 5. This observation led us to introduce the following concept:

Definition 2.4 A *Hardware Machine* (HM) is a pair H=(M, R), where M is an FSM and $R \subseteq WS(M)$ is closed under transition; R is called the set of *operation states* of H.

In the above definition, R must be understood as a set of states into which a ws-sequence ρ brings H from any state. We could have chosen to make ρ a part of definition of an

HM. We will see below that R can be defined as a set of constraints on the boundaries of component slices of a suitable decomposition of M, thus we found it more natural to use R than ρ as part of the HM definition.

3. Post-reboot equivalence

In this section, we introduce post-reboot bisimulation (PRB) and post reboot equivalence, and construct the lattice of PRBs. We show how PRB is related to alignability and discuss how the two differ (theoretically), by defining an upper-semi-lattice on ws-sequences. We also relate PRB with FSM bisimulation as defined in [AGM01].

The following concept of bisimulation for compatible FSMs is induced by the bisimulation concept for LTSs [Par81, Mil89]. Recall that an FSM can be viewed as an LTS by considering a pair $(a, \lambda(s,a))$ as the label for transition $s \rightarrow \delta(s,a)$, where a is an input [HS96].

Definition 3.1: Let $M_i = (S_i, \Sigma, \Gamma, \delta_i, \lambda_i)$, i = 1,2, be FSMs, and let $B \subseteq S_1 \times S_2$ be a relation such that:

- $B(s_1,s_2) \Rightarrow \forall a \in \Sigma$: $\lambda_1(s_1,a) = \lambda_2(s_2,a)$ & $B(\delta_1(s_1,a),\delta_2(s_2,a))$.

Then, B is called an FSM *bisimulation* and M_1 and M_2 are called *bisimilar* with respect to B. States $(s_1,s_2) \in S_1 \times S_2$ are called *bisimilar*, written as $s_1 \sim s_2$, if they are contained in a bisimulation on $S_1 \times S_2$.

State equivalence is an early form of FSM-bisimulation: $s_1 \sim s_2$ iff $s_1 \approx s_2$. Therefore, the state equivalence relation StateEq $\subseteq S(M_1) \times S(M_2)$ is the largest (w.r.t. $\subseteq$) bisimulation between M_1 and M_2. It in interesting to note that FSMs M_1 and M_2 are *replaceable*, or *FSM-equivalent* [Koh78], iff StateEq(M_1,M_2) is a non-empty bisimulation.

Corollary 3.1 Let M_1 and M_2 be FSMs, let $S_1 \subseteq S(M_1)$ and $S_2 \subseteq S(M_2)$ be closed under transition, and let StateEq(S_1,S_2) = StateEq$\cap(S_1 \times S_2)$. Then StateEq(S_1,S_2) is the largest bisimulation between M_1 and M_2 contained in $S_1 \times S_2$.

Definition 3.2 Let $M_i = (S_i, \Sigma, \Gamma, \delta_i, \lambda_i)$, i = 1,2, be FSMs, let π be a sequence of inputs from Σ, and let $B \subseteq S_1 \times S_2$ be a bisimulation between M_1 and M_2. A pair (π,B) is a *post-reboot bisimulation* (or *PRB*) between M_1 and M_2 iff:

- $\forall (s_1,s_2) \in S_1 \times S_2$. ($\pi$: $s_1 \rightarrow^* t_1$ & π: $s_2 \rightarrow^* t_2$) $\Rightarrow B(t_1,t_2)$.

M_1 and M_2 are called *post-reboot bisimilar* or *post-reboot equivalent* if there is a PRB between them.

Theorem 3.1 The set of all post-reboot bisimulations between FSMs M_1 and M_2, when it is a non-empty set, is a complete lattice w.r.t. the partial order defined by: $(\pi_1,B_1) \leq (\pi_2, B_2)$ iff $B_1 \subseteq B_2$.

Let (π,B) be a PRB between M_1 and M_2; then one can associate with π a smallest bisimulation B_π such that (π,B_π) is a PRB; B_π is the intersection of all B_i such that (π, B_i) is a PRB. Therefore, we can define a (strict) order on input sequences as follows: $\pi_1 \prec \pi_2$ iff $B_{\pi 1} \supset B_{\pi 2}$; that is, π_1 cannot transfer all state pairs of $M_1 \times M_2$ into $B_{\pi 2}$, whereas π_2 can; therefore, we call π_2 a *stronger* reboot sequence than π_1. We write $\pi_1 \equiv \pi_2$ iff $B_{\pi 1} = B_{\pi 2}$. When $M_1 = M_2$, the order $\prec$ is in fact an order on the ws-sequences of M_1. The order $\prec$ has upper bounds but need not have a bottom element, thus need not be a lattice. Indeed, consider the FSM $M_{\therefore}$ in Figure 1. Note that $s_1 \approx s_4$ and $s_2 \approx s_3$. Therefore, input sequences 1 and 0 are both ws-sequences of $M_{\therefore}$ and so is 10. Note that $1 \equiv 11 \equiv 111 \ldots$; $0 \equiv 00 \equiv 000 \ldots$; and $01 \equiv 10 \equiv 100 \ldots$. Thus, quotient $\prec / \equiv$ has three elements in $M_{\therefore}$, ordered as in Figure 1.a.

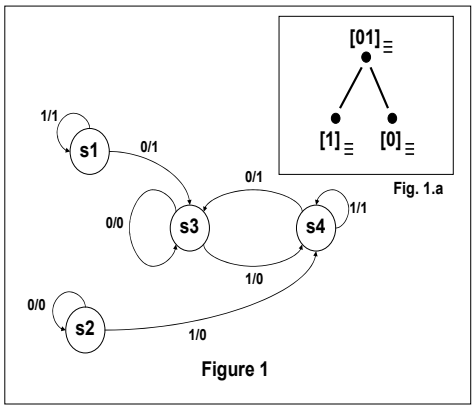


Figure 1

The deep analysis of weak synchronization via *strongly connected components* (or SCCs) of the state transition graphs presented in [PR96] is closely related to our lattice-theoretic characterization of PRB: Recall that an SCC is a set of states where between any two states there is a state transition path. A *sink* SCC is one from which there is no exiting transition. Then, in any smallest PRB (π, B), states in B belong to sink SCCs of M_1 and M_2 and π is a strongest ws-sequence. This follows easily from the construction of ws-sequences that bring any state into a sink SCC, in [PR96], and from Corollary 3.1.

Post-reboot bisimulation relates to the bisimulation concept for FSMs with start states [AGM01] as follows. If an FSM M comes with a start state $s^I \in S(M)$, we assume that there is an input sequence π^I, often of length 1, such that $\forall s \in S(M)$. π^I:$s \rightarrow^* s^I$; that is, π^I is a synchronizing sequence for M. Such an assumption is natural for hardware FSMs, since hardware can power up at any state, and assumption of a start state actually implies assumption of a ws-sequence that brings the FSM into the start state. Now let M_1 and M_2 be FSMs with start states s^I_1 and s^I_2, respectively. Then [AGM01] requires (s^I_1,s^I_2) to belong to any bisimulation $B \subseteq S_1 \times S_2$, which is equivalent to requiring that (π^I,B) is a PRB for any non-empty bisimulation $B \subseteq S_1 \times S_2$, where π^I is such that $\forall s_1 \in S_1$. π^I: $s_1 \rightarrow^* s^I_1$ and $\forall s_2 \in S_2$. π^I: $s_2 \rightarrow^* s^I_2$; but any non-empty bisimulation B should anyway contain (s^I_1,s^I_2) because B is closed under transition and π^I transfers every state pair into (s^I_1,s^I_2).

Definition 3.2 Let M_1 and M_2 be FSMs. Let $S_1 \subseteq S(M_1)$ and $S_2 \subseteq S(M_2)$. We call a bisimulation $B \subseteq S(M_1) \times S(M_2)$ an *on-to bisimulation on $S_1 \times S_2$* iff $B \subseteq S_1 \times S_2$ and

- $\forall s_1 \in S_1.\ \exists s_2 \in S_2.\ B(s_1,s_2)$ & $\forall s_2 \in S_2.\ \exists s_1 \in S_1.\ B(s_1,s_2)$.

Post-reboot equivalence is related to alignability as follows.

Theorem 3.2 Let WS_1 and WS_2 be weak-synchronization states of FSMs M_1 and M_2, respectively ($WS_1 = \emptyset$ if M_1 is not weakly synchronizable, and similarly for WS_2 and M_2). Further, let $StateEq(WS_1,WS_2) = StateEq \cap (WS_1 \times WS_2)$. Then the following are equivalent:

1. M_1 and M_2 are alignable;
2. $StateEq(WS_1,WS_2) \neq \emptyset$;
3. $StateEq(WS_1,WS_2)$ is a non-empty on-to bisimulation, on $WS_1 \times WS_2$, between M_1 and M_2.
4. M_1 and M_2 are post-reboot equivalent.

It is easy to see that in any PRB (π, B) between M_1 and M_2, π is a universal aligning sequence for M_1 and M_2, thus $\prec$ orders universal aligning sequences of M_1 and M_2.

4. Combinational equivalence as post-reboot equivalence

In this section, we examine combinational verification of state-matching FSMs, expose the verification holes in its current formalism, and relate it to post-reboot equivalence in an attempt to come up with a satisfactory formalism.

To do this, we explain in an example the compositional alignability verification framework proposed in [KSKH04]. The specification and implementation FSMs M_1 and M_2 are decomposed into components, as in Figure 2 below.

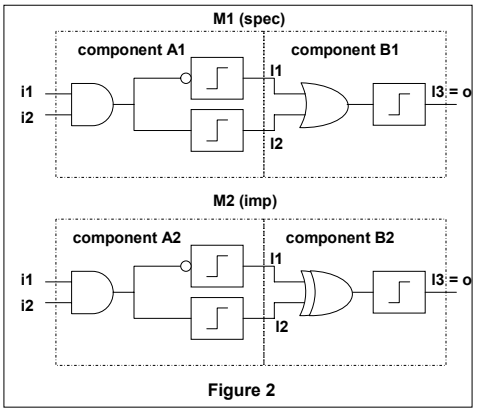


Figure 2

A 1-1 correspondence between the component slices of M_1 and M_2 is defined via mapping the corresponding slice boundaries, where the boundary signals are latches. For example, the boundary signals of components A_1 and A_2 are mapped (they have the same names), and so are the boundary signals of B_1 and B_2. FSM decomposition is needed to reduce equivalence verification for M_1 and M_2 to equivalence verification of the components, and for this to work, boundary properties are added to the components to eliminate behaviors of the components that will never happen during a post-reboot behavior of the FSMs. For example, by using the constraint $l_1 = \neg l_2$, it is possible to prove that the *conditional* FSMs obtained from B_1 and B_2 by restricting the allowed input sequences are alignable. To make usage of such a constraint sound, one must insure that the constraint is valid in all post-reboot operation states. Indeed, in this example it is enough to use any ws-sequence as a reboot sequence to ensure the constraint. Since the constraint is actually the output constraint for components A_1 and A_2, its validity in all post-reboot states of M_1 and M_2 is proved locally in the components A_1 and A_2 – the constraint is valid in all post-ws states of A_1 and A_2.

There is another condition for a safe usage of boundary properties – the resulting conditional FSMs must be stable [KSKH04]. The intuition is that, in stable conditional FSMs, an input vector is allowed in a state transition path iff it is allowed in all state transition paths. Such a conditional FSM can be mapped to an equivalent (non-conditional) FSM whose input signature is a subset of that of the conditional FSM, therefore, the alignability theory is valid for stable conditional FSMs. Components B_1 and B_2 constrained with $l_1 = \neg l_2$ are stable conditional FSMs, because input vectors $l_1 = l_2 = 0$ and $l_1 = l_2 = 1$ are never allowed while the remaining two input vectors are always allowed.

Only a subset of boundary properties is used for the assume-guarantee compositional proofs. Such properties are called *verification properties*. Their conjunction is denoted by VP_D. A decomposition D of (M_1, M_2) is called *stable* if all the components are stable under the verification properties, and the output properties of each component are valid in ws-states of the component (constrained by respective input properties). When D is stable, VP_D determines a relation $R_D \subseteq S(M_1) \times S(M_2)$ as follows: $(s_1,s_2) \in R_D$ iff (s_1,s_2) satisfies VP_D and the induced state of each component in D is a ws-state for that component (constrained with VP_D). It is assumed that the same name (mapped) latches are assigned the same values in (s_1,s_2).

Now recall that two FSMs are called *state-matching* if there is a 1-1 mapping between their latches. Often, for state-matching FSMs, it is allowed that a latch in one model is mapped to more than one latch in the other model. Sometimes, a mapping may have a *polarity*: a latch in one model may be mapped on a negation of a latch in the other model. This slightly more general treatment can easily be reduced to a situation where the latch mapping is 1-1, and no polarity is involved, and we adopt these assumptions.

When performing combinational equivalence verification, a mismatch in functionality of a component pair is allowed if the supporting components (or the supporting logic) in M_1 and M_2 can never generate a value combination that produces the mismatch. For example, the outputs l_1 and l_2 of components A_1 and A_2 of FSMs M_1 and M_2 in Figure 2 can only generate values satisfying the "mutex" property $l_1 = \neg l_2$ (except the start time 0 when the

latches l_1 and l_2 may have arbitrary values). To formalize this intuition, let us call a decomposition D of $M_1 \times M_2$ *combinational* if D is stable and each latch of M_1 and M_2 is an output of a component of M_1 or M_2.

Definition 4.1 Let M_1 and M_2 be state-matching FSMs. We call M_1 and M_2 *combinationally equivalent* if there is a combinational decomposition D of (M_1, M_2) such that:

- Outputs of any matching component pair (A_1,A_2) in D are equal at the next time if their inputs satisfy VP_D at the current time and their outputs are equal at the current time.
- The verification properties on outputs of any component pair (A_1,A_2) in D are valid at the next time if their inputs satisfy VP_D at the current time and their outputs are equal at the current time.

As already explained on an example above, by allowing the properties on component boundaries, one actually assumes a reboot sequence, and expects the FSMs' behaviors to match *post-reboot*. Therefore, unless $M_1 \times M_2$ can be driven from an arbitrary state into a state satisfying R_D, combinational equivalence w.r.t. D is *meaningless* – it is vacuous. Hence the following definition:

Definition 4.2 Let M_1 and M_2 be combinationally equivalent w.r.t. a combinational decomposition D, and let there be an input sequence π that brings any state pair of $M_1 \times M_2$ into a state pair in R_D. Then we call M_1 and M_2 *post-reboot combinationally equivalent* (w.r.t. D).

The problem of verifying that a reboot sequence will bring the circuits into a bisimulation has not been addressed by formal methods and traditionally this was not considered as part of combinational equivalence verification, which is a verification gap. In Section 5, we will discuss how to formally verify whether a pair (π, B) is a PRB, implying that π is a legal reboot sequence for bisimulation B.

Theorem 4.1 State-matching FSMs M_1 and M_2 are combinationally equivalent iff there is a combinational decomposition D of (M_1,M_2) such that R_D is a bisimulation.

Note that, for a combinational decomposition D, if and only if R_D is a bisimulation, every component pair has matching equivalent ws-states, and thus is alignable (by the Alignment Theorem). Hence, we conclude from Theorem 4.1 that state-matching FSMs M_1 and M_2 are combinationally equivalent w.r.t. a combinational decomposition D iff all component pairs of D (constrained by VP_D) are alignable. Finally, we show that post-reboot combinational equivalence is a PRB.

Theorem 4.2 Post-reboot combinationally equivalent state-matching FSMs M_1 and M_2 are post-reboot equivalent.

As already mentioned, combinational verification is often combined with retiming verification on the same design. For weakly synchronizable FSMs, steady-state equivalence, used as the semantics for retiming verification, implies alignability [KH03], thus post-reboot equivalence. Therefore, the retiming verification with steady-state semantics can be safely used as part of compositional post-reboot equivalence verification, provided the retimed components are first proven to be weakly-synchronizable.

5. Verification of Hardware Machines

Now, we consider verification of hardware machines in a wider context, where equivalence verification is combined with assertion verification and reboot sequence validation. We do not intend to cover all methodological aspects; however, we demonstrate the advantages and adequacy of post-reboot equivalence for this task.

Definition 5.1 We call HMs $H_1=(M_1,R_1)$ and $H_2=(M_2,R_2)$ e*quivalent* if there is a post-reboot bisimulation (π,B) between FSMs M_1 and M_2 such that $B \subseteq R_1 \times R_2$. We call a CTL* formula [CGP99] *valid* in H_1 iff it is valid in all states in R_1.

Consider the following FSM M_{PR}, taken from [PR96]. (We use the original notation for states: A,B,C, etc.)

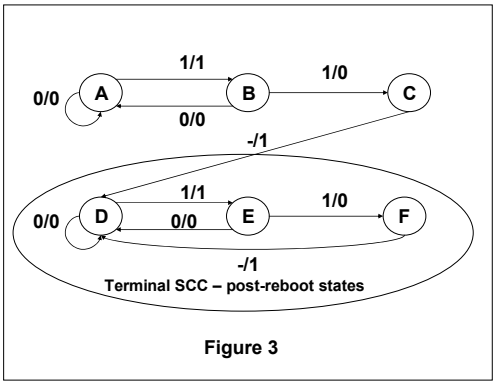


Figure 3

State pairs (A, D), (B, E) and (C, F) are equivalent. Since $A \approx D$, sequence 0 is a ws-sequence for M_{PR}, bringing states {A, B} into A and states {C, D, E, F} into D. Since all states are accessible from A and D, all states are ws-states. Clearly, one has $0 \equiv 01 \equiv 010 \prec 011 \prec 0111 \equiv 0110$. If the designer wants M_{PR} to operate in states {D, E, F} after reboot, he/she can choose a strongest reboot sequence, e.g. 0111, which transfers any state into the sink SCC {D, E, F}. Assume P is a property that is not valid at state C and is valid in {D, E, F}. P can be seen as a behavioral specification for a design that the designer wants to implement. Then, M_{PR} meets its behavioral specification with respect to the post-reboot semantics (for the reboot sequence 0111, for instance) but it does not meet its behavioral specification relative to the alignability semantics (e.g. when 0 is chosen as the ws-sequence). In this respect, adopting post-reboot semantics has a

significant impact on the verification methodology: since alignability does not distinguish between ws-sequences, adopting the alignability semantics forces the designer to modify the implementation of M_{PR} because of a "failing" property P. Here is the data from a chip design project that we supported: 75% of the logic bugs discovered by post-reboot simulation, after the equivalence verification was completed, were caused by the initialization issues.

For an FSM M, adopting the alignability semantics implies the need to prove for a property P the derived property $s \in WS(M) \Rightarrow P(s)$ for any state s of M. On the other hand, adopting post-reboot equivalence for a Hardware Machine (M,R) implies instead proving property $s \in R \Rightarrow P(s)$. We have seen that relation R can be computed (and is computed in practice) as the relation R_D associated to a stable decomposition of M (or M×M), while WS(M) cannot be computed for industrial designs, and it is unclear how a property $WS \Rightarrow P$ can be verified in general for a weakly-synchronizable FSM.

Let P be a CTL* formula written using the common (i.e., mapped) variables of M_1 and M_2 as the atomic propositions; such a formula is observable in both FSMs. Note that when we build a PRB (ρ,R_D) based on a stable decomposition D of (M_1,M_2), the HMs (M_1,R_1) and (M_2,R_2) are equivalent, where R_1 and R_2 are the projections of R_D on $S(M_1)$ and $S(M_2)$, respectively. Further, P is valid in (M_1,R_1) iff it is valid in (M_2,R_2) (cf. [Ch. 11, CGP99]). However, based on $M_1 \approx_{aln} M_2$ alone, it is no longer possible to infer the validity of P on the ws-states of M_2 from its validity on ws-states of M_1. Indeed, let M'_{PR} be the same FSM as the FSM M_{PR} in Figure 3, except now on input 0 the states A and B transition to D (rather than its equivalent state A). Then $M'_{PR} \approx_{aln} M_{PR}$ and {D, E, F} is the set of ws-states of M'_{PR} and, therefore, the property P (from the same example) is valid in all ws-states of M'_{PR} while it is not valid in the ws-state C of M_{PR}. That is, alignability equivalence for FSMs is not informative enough to allow inferring common observable properties from one model to its equivalent model. Note that considering the boundary latches as outputs and thereby strengthening alignability equivalence (by ensuring that the atomic propositions have the same values in M_1 and M_2) does not help us resolve the validity preservation problem with alignability, because the boundary properties are also used in proving the common observable properties.

In Tables 1 and 2, we present information on the verification of 5 assertions on the specification model. In both experiments, the boundaries of the cones on which the properties are checked are built using mapped latches (at this point, the equivalence of corresponding components of specification and implementation is already proven using the verification properties). If verification fails because of a spurious counter-example, the cone is expanded and verification is rerun. The use of the (boundary) verification properties as assumptions is allowed in the experiment in Table 1 and not allowed in the experiment in Table 2. In both cases, a SAT-based initialization algorithm is used to weakly synchronize the cones [RH02]. Thus, the first experiment closely corresponds to post-reboot equivalence verification with respect to the post-reboot bisimulation defined by the stable decomposition employed in the equivalence verification. Since we weakly synchronize the cones before verifying the properties, the second experiment is only an approximation of proving $WS \Rightarrow P$, because the computed synchronization sequence may reset the cone into a proper subset of the WS states. (We do not know how to prove $WS \Rightarrow P$ for large FSMs.)

In the tables, we present the highest level of expansion iterations (EI), the size of the cones, the number of boundary properties (BP) used in assertion verification, and the runtimes. All properties in Table 1 are verified using a SAT-based model checker, whereas the same properties in Table 2 cannot be verified. Some of them cannot be verified because of failures to weakly synchronize the cones, and some because of the resulting spurious counter-examples. As expected, the use of boundary properties as assumptions helps confine verification to smaller cones.

assertions	EI	inputs	latches	gates	BP	cpu (sec)
assert 1	2	114	26	1674	17	154
assert 2	1	24	20	221	1	155
assert 3	0	18	3	1012	9	209
assert 4	0	835	4	6986	64	209
assert 5	0	19	4	873	2	312

Table 1: assertion verification using boundary properties

assertions	EI	inputs	latches	gates	cpu (sec)
assert 1	10	125	526	7871	256
assert 2	10	369	1133	17690	262
assert 3	9	69	114	954	211
assert 4	2	858	37424	250745	596
assert 5	1	675	18	2585	1325

Table 2: assertion verification without boundary properties

Now let us turn to the question of verifying whether or not a sequence π is a ws-sequence for an FSM M. Since it is impossible to symbolically simulate [Jon02] a full-chip design, an over-approximation of states into which a reboot sequence brings a design is computed using 3-valued simulation, often also called X-simulation [HC98]. At the beginning of simulation, all latch values are set to the X value, and all inputs except a few reset signals are simulated with X. Propagation of reset signal values forces assignment of binary values to most of the latches, and the first part of the reboot sequence ends as soon as a fix-point of simulation is reached. Reboot sequence then continues to initialize counters, memory addresses, and other latches to specific values. To the best of our knowledge, no *practical, formal* methods exist for checking whether the over-approximation set of states of a combined full-chip design

$M_1 \times M_2$, built, as above, using 3-valued simulation, is indeed a subset of ws-states of $M_1 \times M_2$. Such a method would involve model checking on a huge state space.

The above question is actually irrelevant for achieving compositional post-reboot equivalence verification. Suppose a stable decomposition D of $M_1 \times M_2$ has been built. Let S be the 3-valued state obtained by simulating $M_1 \times M_2$ with π, starting from state X, let OS be the set of (binary) states of $M_1 \times M_2$ induced by S (latches with X values are assigned all possible binary value combinations), and let OS^* denote the closure of OS under the transition relation. Then it is enough to prove that (π, OS^*) is a PRB. The latter is a model-checking problem *on the components* of the decomposition (which are within the capacity of the model checker): for each component pair (A_1,A_2) of $M_1 \times M_2$, the state pair (t_1,t_2) induced by any state of OS must be an equivalence state of $A_1 \times A_2$, and the verification properties on the outputs of (A_1,A_2) must be valid for (t_1,t_2) and any state pair reachable from it by any input sequence of (A_1,A_2) satisfying its input properties.

6. Conclusions

We have proposed a new, finer formalism for modeling hardware – Hardware Machines, where the set of post-reboot operation states is a key component of the definition. This led us to introduce post-reboot equivalence, where, unlike alignability equivalence, the operation states play an important role in the semantics. Indeed, we could refine the alignability equivalence into a complete lattice of post-reboot bisimulations, and refine the homogeneous class of ws-sequences into an upper semi-lattice of reboot sequences. This new view of hardware also led us to a revision of the existing widespread equivalence concepts and the way they are employed in practice. We gain a new insight into compositional hardware verification, where the construction of a set of operation states is a by-product of building a stable decomposition of specification and implementation models. As a result of this revision, we were able to point to verification gaps in the existing methods, and propose a unified theory that bridges the verification practice to the Hardware Machine formalism.

We have briefly touched on the subject of assertion verification for Hardware Machines, demonstrating that the shift from FSMs to Hardware Machines implies important differences in the semantics of temporal logic assertions. We presented experimental evidence on how such a change in assertion semantics affects assertion verification in practice. We leave it to future work to come up with a comprehensive assertion verification theory and methodology that will be fully aligned with compositional equivalence verification and reboot sequence verification.

For non-state-matching designs, there are too many options to decompose the design into sub-circuits, and, at present, building stable decompositions is semi-automatic: heuristic latch mapping algorithms are used to define decomposition to start with, and then abstraction refinement methods [CGP99] are used to adjust the sub-circuit boundaries and add properties. Defining a fully automatic algorithm for building stable decompositions is a challenging direction for future work.

Acknowledgements: Without understanding a complete picture of design verification in practice, this work would have been impossible. We thank S. Goldenberg, R. Kaivola, M. Mneimneh, and A. Jas for the numerous discussions that oriented us in our search for a "right equivalence concept" for full-chip verification. We would also like to thank N. Piterman, A. Pnueli and M. Vardi for their valuable feedback that helped us sharpen the concepts.

References

[AGM01] Ashar P., A. Gupta, S. Malik, Using complete-1-destinguishability for FSM equivalence checking, ACM Trans. on Design Automation of Electronic Sys, vol. 6, no. 4, 2001.

[CGP99] Clarke E.M., O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999.

[DP90] Davey, B.A., Priestley H.A., *Introduction to Lattices and Order*, Cambridge University Press, 1990.

[HC98] Huang, S.-Y., K.-T., Cheng, *Formal Equivalence Checking and Design Debugging*, Kluwer, 1998.

[HS96] Hachtel G.D., F. Somenzi, *Logic Synthesis and Verification Algorithms*, Kluwer, 1996.

[Jon02] R.B. Jones, R.B. Symbolic Simulation Methods for Industrial Formal Verification, Kluwer, 2002.

[KH02] Khasidashvili Z, Z. Hanna, TRANS: Efficient sequential verification of loop-free circuits, HLDVT02.

[HH03] Khasidashvili, Z., Z. Hanna, SAT-Based methods for sequential hardware equivalence verification without synchronization, BMC'03, ENTCS 89 (4), 2003

[KSKH04] Khasidashvili Z., M. Skaba, D. Kaiss, Z. Hanna, Theoretical framework for compositional sequential hardware equivalence verification in presence of design constraints, ICCAD04, 2004.

[KvE04] Kuehlmann A., C.A.J van Eijk, Combinational and sequential equivalence checking, in: Logic Synthesis and Verification, Kluwer Academic Publishers, 2004.

[Koh78] Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill, 1978.

[Mil89] Milner, A.J.R.G. *Communication and Concurrency*, Prentice Hall, 1989.

[LS91] Leiserson C. E., J. B. Saxe. *Retiming synchronous circuitry*. Algorithmica, 6(1), 1991.

[Par81] Park, D. Concurrency and automata on infinite sequences, 5th GI-Conf. on TCS, LNCS 104, 1981.

[Pix92] Pixley, C. A theory and implementation of sequential hardware equivalence, IEEE trans. CAD, 1992.

[PR96] Pomeranz, I., S.M. Reddy, On removing redundancies from synchronous sequential circuits with synchronizing sequences, IEEE Trans. Computers, 1996.

[RSSB99] Ranjan R.K., V. Singhal, F. Somenzi, R.K. Brayton, Combinational verification for sequential circuits, DATE 1999.

[RH02] Rosenmann, A., Z. Hanna, Alignability equivalence of synchronous sequential circuits, HLDVT '02.

[SPAB01] Singhal, V., C. Pixley, A. Aziz, and R.K. Brayton, Theory of safe replacement for sequential circuits, IEEE Trans. on CAD of integrated circuits and systems, vol. 20, n.2, 2001.

Synchronous Elastic Networks

Sava Krstić
Strategic CAD Labs, Intel Corporation
Hillsboro, Oregon, USA

Jordi Cortadella
Universitat Politècnica de Catalunya
Barcelona, Spain

Mike Kishinevsky, John O'Leary
Strategic CAD Labs, Intel Corporation
Hillsboro, Oregon, USA

***Abstract*— We formally define—at the stream transformer level—a class of synchronous circuits that tolerate any variability in the latency of their environment. We study behavioral properties of networks of such circuits and prove fundamental compositionality results. The paper contributes to bridging the gap between the theory of latency-insensitive systems and the correct implementation of efficient control structures for them.**

I. Introduction

The conventional abstract model for a synchronous circuit is a machine that reads inputs and writes outputs at every cycle. The outputs at cycle i are produced according to a calculation that depends on the inputs at cycles $0, \ldots, i$. Computations and data transfers are assumed to take zero delay.

Latency-insensitive design by Carloni et al. [2] aims to relax this model by elasticizing the time dimension and so decoupling the cycles from the calculations of the circuit. It enables the design of circuits tolerant to any discrete variation (in the number of cycles) of the computation and communication delays. With this modular approach, the functionality of the system only depends on the functionality of its components and not on their timing characteristics.

The motivation for latency-insensitive design comes from the difficulties with timing and communication in nanoscale technologies. The number of cycles required to transmit data from a sender to a receiver is governed by the distance between them, and often cannot be accurately known until the chip layout is generated late in the design process. Traditional design approaches require fixing the communication latencies up front, and these are difficult to amend when layout information finally becomes available. Elastic circuits offer a solution to this problem. In addition, their modularity promises novel methods for microarchitectural design that can use variable-latency components and tolerate static and dynamic changes in communication latencies, while—unlike asynchronous circuits—still employing standard synchronous design tools and methods.

Cortadella et al. [4] present a simple elastic protocol, called SELF (Synchronous Elastic Flow) and describe methods for efficient implementation of elastic systems and for conversion of regular synchronous designs into elastic form. Inspired by the original work on latency-insensitive design [2], SELF also differs from it in ways that render the theory developed in [2] hardly applicable.

In this paper we give theoretical foundations of SELF: a novel and arguably more practicable definition of elasticity, and the basic compositionality results. For space reasons, the proofs are omitted, but are available in the technical report [7].

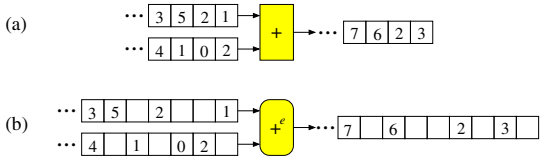


Fig. 1. (a) Conventional synchronous adder, (b) Synchronous elastic adder.

A. Overview

Figure 1(a) depicts the timing behavior of a conventional synchronous adder that reads input and produces output data at every cycle (boxes represent cycles). In this adder, the i-th output value is produced at the i-th cycle. Figure 1(b) depicts a related behavior of an elastic adder—a synchronous circuit too—in which data transfer occurs in some cycles and not in others. We refer to the transferred data items as *tokens* and we say that idle cycles contain *bubbles*.

Put succinctly, elasticization decouples cycle count from token count. In a conventional synchronous circuit, the i-th token of a wire is transmitted at the i-th cycle, whereas in a synchronous elastic circuit the i-th token is transmitted at some cycle $k \geq i$.

Turning a conventional synchronous adder into a synchronous elastic adder requires a communication discipline that differentiates idle from non-idle cycles (bubbles from tokens). In SELF, this is implemented by a pair of single-bit control wires: *Valid* and *Stop*. Every input or output wire Z in a synchronous component is associated to a *channel* in the elastic version of the same component. The channel is a triple of wires $\langle Z, \mathsf{valid}_Z, \mathsf{stop}_Z \rangle$, with Z carrying the data and the other two wires implementing the control bits, as shown in Figure 2(b). A token is transferred on this channel when $\mathsf{valid}_Z \wedge \neg \mathsf{stop}_Z$: the sender sends valid data and the receiver is ready to accept it; see Figure 4. Additional constraints that guarantee correct elastic behavior are given in Section III. There we define precisely the class of elastic circuits and what it means for a circuit A^{e} to be an elastization of a given circuit A. In particular, our definition implies liveness: A^{e} produces infinite streams of tokens if its environment produces infinite streams of tokens at the input channels and is ready to accept infinite streams at the output channels.

Suppose $\mathcal{N}$ is a network of standard (non-elastic) components, as in Figure 2(a). Suppose we then take elasticizations of

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

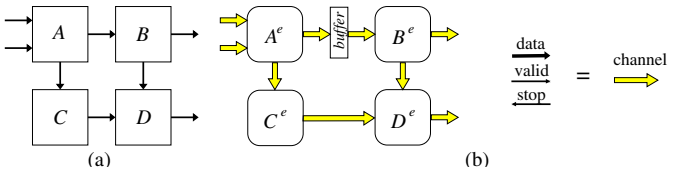


Fig. 2. A synchronous network (a) and its elastic counterpart (b).

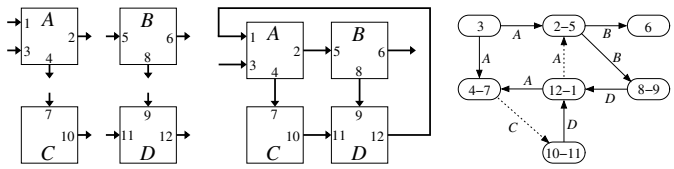


Fig. 3. Four machines (left) put into a network $\mathcal{N}$ (middle), and the network's dependency graph $\Delta(\mathcal{N})$ (right). The nodes of $\Delta(\mathcal{N})$ are wires; internal wires get two labels. The arcs are *non-sequential* input-output wire pairs of component circuits. Dotted arcs indicate that (1,2) and (7,10) are sequential pairs for A and C resp.; they are not part of $\Delta(\mathcal{N})$ so $\Delta(\mathcal{N})$ is acyclic.

these standard components and join their channels accordingly, as in Figure 2(b), ignoring the buffer. Will the resulting network $\mathcal{N}^{\mathrm{e}}$ be an elasticization of $\mathcal{N}$? Will it be elastic at all? These fundamental questions are answered by Theorem 4 of Section IV, which is the main result of the paper. The answers are "yes", provided a certain graph $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}})$ associated with $\mathcal{N}^{\mathrm{e}}$ is acyclic. This graph captures the information about paths inside elastic systems that contain no tokens—analogous to combinational paths in ordinary systems. Importantly, $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}})$ can be constructed using only local information (the "sequentiality interfaces") of the individual elastic components.

Since elastic networks tolerate any variability in the latency of the components, empty FIFO buffers can be inserted in any channel, as shown in Figure 2(b), without changing the functional behavior of the network. This practically important fact is proved as a consequence of Theorem 4.

Synchronous circuits are modeled in this paper as stream transformers, called *machines*. This well-known technique (see [8] and references therein) appears to be quite underdeveloped. Our rather lengthy preliminary Section II elaborates the necessary theory of networks of machines, culminating with a surprisingly novel combinational loop theorem (Theorem 1).

Figure 3 illustrates Theorem 1 and, by analogy, Theorem 4 as well. It relies on the formalization of the notion of combinational dependence at the level of input-output wire pairs. Each input-output pair of a machine is either *sequential* or not, and the set of sequential pairs provides a machine's "sequentiality interface". When several machines are put together into a network $\mathcal{N}$, their sequentiality interfaces define the graph $\Delta(\mathcal{N})$, the acyclicity of which is a test for the network to be a legitimate machine itself.

Elasticizations of ordinary circuits are not uniquely defined. On the other hand, for every elastic machine A there is a unique standard machine, denoted A^{T}, that corresponds to it. We do not discuss any specific elasticization procedures in this paper, but state our results in the form that only involves elastic machines and their unique standard counterparts. This makes the results applicable to multiple elasticization procedures.

B. Related Work

Carloni et al. [2] pioneered a theory of latency-insensitive circuits based on their notion of *patient processes*. Patient processes are defined at a high level of abstraction that models communication on a channel only by "token or bubble", leaving implementation protocol(s) unspecified. In the companion paper [3], Carloni et al. give an incomplete description of an implementation protocol. Assuming our recovery of that protocol (let us call it LID) is accurate, its transfer condition is more complex than that of SELF (Figure 4) and consequently LID requires significantly more complex implementation. For example, conversion of a regular design into LID form needs a wrapper or registers around every module, increasing the latency of each module's computation by two cycles—a penalty that is not required in the SELF elasticization. There might also be practical challenges in interfacing a LID system with an existing non-LID module, requiring the latter to generate stop signals with complex semantics.

cycle	0	1	2	3	4	5	6	7	8	9	...
data_Z	$*$	A	**B**	B	B	C	$*$	$*$	D	D	...
valid_Z	0	1	1	1	1	1	0	0	1	1	...
stop_Z	0	0	1	1	0	0	0	1	1	0	...
SELF	□	t	□	□	t	t	□	□	□	t	...
LID	□	t	t	□	t	t	□	□	□	t	...

Fig. 4. Comparing the SELF and LID protocols. The bottom rows show the states of the channel Z, differentiating between bubbles (□) and tokens (t). When $\neg\mathsf{valid}_Z$, the value at the data wire is irrelevant (labelled $*$ in cycles 0, 6 and 7). The receiver can issue a stop_Z even when the sender does not send valid data (cycle 7). In the cycles 3, 4, and 9, the sender persistently maintains the same valid data as in the previous cycle. In SELF, data transfer takes place in cycles 1,4,5,9, so the transferred sequence is $ABCD\ldots$. In LID, the same sequence of values on the channel wires signifies transfer of a different sequence of data: $A\mathbf{B}BCD\ldots$ This is because a token is transferred on the LID channel when $\mathsf{valid}_Z \wedge \neg(\mathsf{stop}_Z \wedge pre(\mathsf{stop}_Z))$, where pre stands for the value during the previous cycle. (The first occurrence of the stop request $\mathsf{stop}_Z = 1$ means "perhaps you will need to stop next cycle" and the data item B sent through the channel during cycle 2 is assumed to be successfully transmitted to the receiver.)

We emphasize that the limitations of LID implementations are not inherent to the concept of patient processes. Regarding latency properties, they do not seem to be more limited than elastic systems. Still, it turns out that patient processes are not general enough to model elastic systems as we define them in Section III. This we prove in Section V where patient processes and elastic systems are compared as alternative formalizations of latency-insensitive circuits.

Suhaib et al. [12] revisited and generalized Carloni's elasticization procedure, validating its correctness by a simulation method based on model checking.

Lee et al. [9] study *causality interfaces* (pairwise input-output dependencies) and are "interested in existence and uniqueness of the behavior of feedback composition", but do not go as far as deriving a combinational loop theorem.

In their work on design of interlock pipelines [6], Jacobson et al. use a protocol equivalent to SELF, without explicitly

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

specifying it.

Manohar and Martin discuss "slack elasticity" of asynchronous implementations in [10]. Their slack elasticity conditions relate to the structure of choices in the asynchronous specification. Unlike [10], in the current paper we deal with synchronous systems and we take a black box view of their control—no information about the control flow (and hence on the structure of choices) is ever used. Instead the connectivity information corresponding to the system data-flow is used for elasticization. Conservatively ignoring control flow may lead to a performance penalty, but simplifies the translation to an elastic system.

II. Circuits as Stream Functions

In this section we introduce *machines* as a mathematical abstraction of *circuits without combinational cycles*. For simplicity, this abstraction implicitly assumes that all sequential elements inside the circuit are *initialized*. Extending to partially initialized systems appears to be trivial. While there is a large body of work studying circuits or equivalent objects with *good (e.g. constructive [1]) combinational cycles* and their composition (e.g. [5]), we deliberately restrict consideration to the fully acyclic objects, since neither logic synthesis nor timing analysis can properly treat circuits with combinational cycles.

Most of the effort in this section goes into establishing modularity conditions guaranteeing that a system obtained as a network of machines (the feedback construction in particular) is a machine itself.

A. Streams

A *stream over* A is an infinite sequence whose elements belong to the set A. The first element of a stream a is referred to by $a[0]$, the second by $a[1]$, etc. For example, the equation $a[i] = 3i + 1$ describes the stream $(1, 4, 7, \ldots)$.

The set of all streams will be denoted A^∞. Occasionally we will need to consider finite sequences too; the set of all, finite or infinite, sequences over A is denoted A^ω.

We will write $a \sim_k b$ to indicate that the streams a and b have a common prefix of length k. The equivalence relations $\sim_0, \sim_1, \sim_2, \ldots$ are progressively finer and have trivial intersection. Thus, to prove two sequences a and b are equal, it suffices to show $a \sim_k b$ holds for every k. Note also that $a \sim_0 b$ holds for every a and b.

We will use the equivalence relations $\sim_k$ to express properties of systems and machines viewed as multivariate stream functions. All these properties will be derived from the following two basic properties of single-variable stream functions $f\colon A^\infty \to B^\infty$.

causality: $\forall a, b \in A^\infty.\, \forall k \geq 0.\ a \sim_k b \Rightarrow f(a) \sim_k f(b)$

contraction: $\forall a, b \in A^\infty.\, \forall k \geq 0.\ a \sim_k b \Rightarrow f(a) \sim_{k+1} f(b)$

Informally, f is causal if (for every a) the first k elements of $f(a)$ are determined by the first k elements of a, and f is contractive if the first k elements of $f(a)$ are determined by the first $k-1$ elements of a.

Lemma 1: If $f\colon A^\infty \to A^\infty$ is contractive, then it has a unique fixpoint.

Remark. One can define the *distance* $d(a, b)$ between sequences a and b to be $1/2^k$, where k is the length of the largest common prefix of a and b. This gives the sets A^∞ and A^ω the structure of complete metric spaces and Lemma 1 is an instance of Banach Fixed Point Theorem. See the review paper [8] for more details and references about the metric semantics of systems and [13] for "diadic arithmetic of circuits". We choose not to use the metric space terminology in this paper since all "metric reasoning" we need can be as easily done with equivalence relations $\sim_k$ instead. See [11] for principles of reasoning with such "converging equivalence relations" in more general contexts.

B. Systems

Suppose W is a set of typed *wires*; all we know about an individual wire w is a set $\mathsf{type}(w)$ associated to it. A *W-behavior* is a function σ that associates a stream $\sigma.w \in \mathsf{type}(w)^\infty$ to each wire $w \in W$. The set of all W-behaviors will be denoted $[\![W]\!]$. Slightly abusing the notation, we will also write $[\![w]\!]$ for the set $\mathsf{type}(w)^\infty$. Notice that the equivalence relations $\sim_k$ extend naturally from streams to behaviors:

$$\sigma \sim_k \sigma' \quad \text{iff} \quad \forall w \in W.\ \sigma.w \sim_k \sigma'.w$$

Notice also that a W-behavior σ can be seen as a single stream $(\sigma[0], \sigma[1], \ldots)$ of *W-states*, where a state is an assignment of a value in $\mathsf{type}(w)$ to each wire w.

Definition 1: A *W-system* is a subset of $[\![W]\!]$.

Example. A circuit that at each clock cycle receives an integer as input and returns the sum of all previously received inputs is described by the W-system $\mathcal{S}$, where W consists of two wires u, v of type $\mathbb{Z}$, and $\mathcal{S}$ consists of all stream pairs $(a, b) \in \mathbb{Z}^\infty \times \mathbb{Z}^\infty$ such that $b[0] = 0$ and $b[n] = a[0] + \cdots + a[n-1]$ for $n > 0$. Each stream pair (a, b) represents a behavior σ such that $\sigma.u = a$ and $\sigma.v = b$.

We will use wires as typed variables in formulas meant to describe system properties. The formulas are built using ordinary mathematical and logical notation, enhanced with temporal operators *next*, *always*, and *eventually*, denoted respectively by $(\_)^+, \mathsf{G}, \mathsf{F}$. As an illustration, the system $\mathcal{S}$ in the example above is characterized by the property $v = 0 \wedge \mathsf{G}\,(v^+ = v + u)$. Also, one has $\mathcal{S} \models \mathsf{F}\,\mathsf{G}\,(u > 0) \Rightarrow \mathsf{F}\,\mathsf{G}\,(v > 1000)$, where $\models$ is used to denote that a formula is true of a system.

C. Operations on Systems

If $W' \subseteq W$, there is an obvious projection map $\sigma \mapsto \sigma \downarrow W'\colon [\![W]\!] \to [\![W']\!]$. These projections are all one needs for the definition of the following two basic operations on systems.

Definition 2: (a) If $\mathcal{S}$ is a W-system and $W' \subseteq W$, then *hiding* W' in $\mathcal{S}$ produces a $(W - W')$-system $\mathsf{hide}_{W'}(\mathcal{S})$ defined by

$$\tau \in \mathsf{hide}_{W'}(\mathcal{S}) \quad \text{iff} \quad \exists \sigma \in \mathcal{S}.\ \tau = \sigma \downarrow (W - W').$$

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

(b) The *composition* of a W_1-system $\mathcal{S}_1$ and a W_2-system $\mathcal{S}_2$ is a $(W_1 \cup W_2)$-system $\mathcal{S}_1 \sqcup \mathcal{S}_2$ defined by

$$\sigma \in \mathcal{S}_1 \sqcup \mathcal{S}_2 \;\text{ iff }\; \sigma \downarrow W_1 \in \mathcal{S}_1 \wedge \sigma \downarrow W_2 \in \mathcal{S}_2.$$

If W and W' are disjoint wire sets, $\sigma \in [\![W]\!]$, and $\tau \in [\![W']\!]$, then there is a unique behavior $\vartheta \in [\![W \cup W']\!]$ such that $\sigma = \vartheta \downarrow W$ and $\tau = \vartheta \downarrow W'$. This "product" of behaviors will be written as $\vartheta = \sigma * \tau$. (If W is the empty set, then $[\![W]\!]$ has one element—a "trivial behavior" that is also a multiplicative unit for the product operation $*$.) We will also use the notation $[u \mapsto a, v \mapsto b, \ldots]$ for the $\{u, v, \ldots\}$-behavior σ such that $\sigma.u = a$, $\sigma.v = b$, etc.

Hiding and composition suffice to define complex networks of systems. To model identification of wires, we use simple *connection systems*: by definition, $\mathsf{Conn}(u, v)$ is the $\{u, v\}$-system consisting of all behaviors σ such that $\sigma.u = \sigma.v$.

Now if $\mathcal{S}_1, \ldots, \mathcal{S}_m$ are given systems and u_1, …, u_n, v_1, …, v_n are some of their wires, the network obtained from these systems by identifying each wire u_i with the corresponding wire v_i (of equal type) is the system

$$\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,|\, u_1 = v_1, \ldots, u_n = v_n \rangle$$

defined as $\mathsf{hide}_{\{u_1,\ldots,u_n,v_1,\ldots,v_n\}}(\mathcal{S})$, where

$$\mathcal{S} = \mathcal{S}_1 \sqcup \cdots \sqcup \mathcal{S}_m \sqcup \mathsf{Conn}(u_1, v_1) \sqcup \cdots \sqcup \mathsf{Conn}(u_n, v_n).$$

The simplest case ($m = n = 1$) of networks is the construction

$$\langle \mathcal{S} \,|\, u = v \rangle = \mathsf{hide}_{\{u,v\}}(\mathcal{S} \sqcup \mathsf{Conn}(u, v)),$$

used for a *feedback* definition in Section II-E. A behavior σ belongs to $\langle \mathcal{S} \,|\, u = v \rangle$ if and only if $\sigma * [u \mapsto a, v \mapsto a] \in \mathcal{S}$ for some $a \in [\![u]\!]$.

D. Machines

Suppose I and O are disjoint sets of wires, called *inputs* and *outputs*, correspondingly. By definition, an (I, O)-*system* is just an $(I \cup O)$-system. Consider the following properties of an (I, O)-system $\mathcal{S}$.

deterministic:

$$\forall \omega, \omega' \in \mathcal{S}.\ \omega \downarrow I = \omega' \downarrow I \;\Rightarrow\; \omega \downarrow O = \omega' \downarrow O$$

functional:

$$\forall \sigma \in [\![I]\!].\, \exists! \tau \in [\![O]\!].\, \sigma * \tau \in \mathcal{S}$$

causal:

$$\forall \omega, \omega' \in \mathcal{S}. \forall k \geq 0.\ \omega \downarrow I \sim_k \omega' \downarrow I \;\Rightarrow\; \omega \downarrow O \sim_k \omega' \downarrow O$$

Clearly, functionality implies determinism. Conversely, a deterministic system is functional if and only if it accepts all inputs. Note also that causality implies determinism: if $\omega \downarrow I = \omega' \downarrow I$, then $\omega \downarrow I \sim_k \omega' \downarrow I$ holds for every k, so $\omega \downarrow O \sim_k \omega' \downarrow O$ holds for every k too, so $\omega \downarrow O = \omega' \downarrow O$.

Definition 3: An (I, O)-*machine* is an (I, O)-system that is both functional and causal.

A functional system $\mathcal{S}$ uniquely determines and is determined by the function $F\colon [\![I]\!] \to [\![O]\!]$ such that $F(\sigma) = \tau$ holds if and only if $\sigma * \tau \in \mathcal{S}$. The causality condition for such $\mathcal{S}$ can be also written as follows:

$$\forall \sigma, \sigma' \in [\![I]\!]. \forall k \geq 0.\ \sigma \sim_k \sigma' \Rightarrow F(\sigma) \sim_k F(\sigma').$$

The system in the example in Section II-B is a machine if we regard u as an input wire and v as an output wire. The same is true of the system $\mathsf{Conn}(u, v)$: its associated function F is the identity function.

E. Feedback on Machines

We will use the term *feedback* for the system $\langle \mathcal{S} \,|\, u = v \rangle$ as mentioned in Section II-C when $\mathcal{S}$ is a machine and the wires u and v of the same type are an input and output of $\mathcal{S}$ respectively. Our concern now is to understand under what conditions the feedback produces a machine.

To fix the notation, assume $\mathcal{S}$ is an (I, O)-machine given by $F\colon [\![I]\!] \to [\![O]\!]$, with wires $u \in I$, $v \in O$ of the same type A. By the note at the end of Section II-C, we have that for every $\sigma \in [\![I - \{u\}]\!]$ and $\tau \in [\![O - \{v\}]\!]$,

$$\sigma * \tau \in \langle \mathcal{S} \,|\, u = v \rangle$$

if and only if

$$\exists a \in A^\infty.\, F(\sigma * [u \mapsto a]) = \tau * [v \mapsto a]),$$

so $\langle \mathcal{S} \,|\, u = v \rangle$ is functional when the function $F^\sigma_{uv} : A^\infty \to A^\infty$ defined by $F^\sigma_{uv}(a) = F(\sigma * [u \mapsto a]).v$ has a unique fixpoint. By Lemma 1, this is guaranteed if F^σ_{uv} is contractive.

The following definition introduces the key concept of sequentiality that formalizes the intutive notion that there is no combinational dependence of a given output wire on a given input wire. Sequentiality of the pair (u, v) easily implies contractivity of F^σ_{uv} for all σ.

Definition 4: The pair (u, v) is *sequential* for $\mathcal{S}$ if for every $\sigma, \sigma' \in [\![I]\!]$ and every $k \geq 0$

$$\begin{array}{l} \sigma.u \sim_{k-1} \sigma'.u \\ \wedge\ \forall x \in I - \{u\}.\ (\sigma.x \sim_k \sigma'.x) \end{array} \;\Rightarrow\; F(\sigma).v \sim_k F(\sigma').v$$

Lemma 2 (Feedback): If (u, v) is a sequential input-output pair for a machine $\mathcal{S}$, then the feedback system $\langle \mathcal{S} \,|\, u = v \rangle$ is a machine too.

Example. Consider the system $\mathcal{S}$ with $I = \{u, v\}$, $O = \{w, z\}$, specified by equations

$$w = u \oplus ((0)\#v) \qquad z = v \oplus v,$$

where all wires have type $\mathbb{Z}$, the symbol $\oplus$ denotes the componentwise sum of streams, and $\#$ denotes concatenation. Since z does not depend on u, the pair (u, z) is sequential. The pair (v, w) is also sequential since to compute a prefix of w it suffices to know (a prefix of the same size of u and) a prefix of smaller size of v. The remaining two input-output pairs (u, w) and (v, z) are not sequential.

To find the machine $\langle \mathcal{S} \,|\, v = w \rangle$, we need to solve the equation $v = u \oplus ((0)\#v)$ for v. For each $u = (a_0, a_1, a_2, \ldots)$, the equation has a unique solution $v = \hat{u} = (a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots)$. Substituting the solution into $z = v \oplus v$, we obtain

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

a description of $\langle \mathcal{S} \,|\, v = w \rangle$ by a single equation that relates its input and output: $z = \hat{u} \oplus \hat{u}$. The other feedback $\langle \mathcal{S} \,|\, u = z \rangle$ is easier to calculate; it is given by equation $w = v \oplus v \oplus ((0) \# v)$.

F. Networks of Machines and the Combinational Loop Theorem

Consider a network $\mathcal{N} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,|\, u_1 = v_1, \ldots, u_n = v_n \rangle$, where $\mathcal{S}_1, \ldots, \mathcal{S}_m$ are machines with disjoint wire sets and the pairs (u_1, v_1),...,(u_n, v_n) involve n distinct input wires u_i and n distinct output wires v_i. (There is no assumption that u_i, v_i belong to the same machine $\mathcal{S}_j$.) Our goal is to understand under what conditions the system $\mathcal{N}$ is a machine.

Note that $\mathcal{N} = \langle \mathcal{S} \,|\, u_1 = v_2, \ldots, u_n = v_n \rangle$, where $\mathcal{S} = \mathcal{S}_1 \sqcup \cdots \sqcup \mathcal{S}_m$. It is easy to check that an input-output pair (u, v) of $\mathcal{S}$ is sequential if either (1) (u, v) is sequential for some $\mathcal{S}_i$, or (2) u and v belong to different machines. Thus, the information about sequentiality of input-output pairs of the "parallel composition" machine $\mathcal{S}$ is readily available from the sequentiality information about the component machines $\mathcal{S}_i$, and our problem boils down to determining when a multiple feedback operation performed on a single machine results in a system that is itself a machine.

Simultaneous feedback specified by a set of two or more input-output pairs of a machine does not necessarily produce a machine even if all pairs involved are sequential. Indeed, in the example in Section II-E, we had a system $\mathcal{S}$ with two sequential pairs (u, z) and (v, w), but (u, z) ceases to be sequential for $\langle \mathcal{S} \,|\, v = w \rangle$. Indeed, if z and u are related by $z = \hat{u} \oplus \hat{u}$, then knowing a prefix of length k of z requires knowing the prefix of the same length of u; a shorter one would not suffice.

To ensure that a multiple feedback construction produces a machine, one needs to show that, in addition to the wire pairs to be identified, sufficiently many other input-output pairs are also sequential. A precise formulation for a *double* feedback is given by a version of the Bekić Lemma: for the system $\langle \mathcal{S} \,|\, u = w, v = z \rangle$ to be a machine, it suffices that *three* pairs of wires be sequential—(u, w), (v, z), and one of (u, z), (v, w). This non-trivial auxiliary result is needed for the proof of Theorem 1 below, and is a special case of it.

Given an (I, O)-machine $\mathcal{S}$, let its *dependency graph* $\Delta(\mathcal{S})$ have the vertex set $I \cup O$ and directed edges that go from u to v for each pair $(u, v) \in I \times O$ that is *not* sequential. For a network system $\mathcal{N} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,|\, u_1 = v_1, \ldots, u_n = v_n \rangle$, its graph $\Delta(\mathcal{N})$ is then defined as the direct sum of graphs $\Delta(\mathcal{S}_1), \ldots, \Delta(\mathcal{S}_m)$ with each vertex u_i $(1 \le i \le n)$ identified with the corresponding vertex v_i (Figure 3).

Theorem 1 (Combinational Loop Theorem): The network system $\mathcal{N}$ is a machine if the graph $\Delta(\mathcal{N})$ is acyclic.

III. Elastic Machines

In this section we give the definition of elastic machines. Its four parts—input-output structure, persistence conditions, liveness conditions, and the transfer determinism condition—are covered by Definitions 5-8 below.

A. Input-output Structure, Channels, and Transfer

We assume that the set of wires is partitioned into *data*, *valid*, and *stop* wires, so that for each data wire X there exist associated wires valid_X and stop_X of boolean type. (In actual circuit implementations, valid_X and stop_X need not be physical wires; it suffices that they be appropriately encoded.)

Definition 5: Let I, O be disjoint sets of data wires. An $[I, O]$*-system* is an (I', O')-machine, where $I' = I \cup \{\mathsf{valid}_X \,|\, X \in I\} \cup \{\mathsf{stop}_Y \,|\, Y \in O\}$ and $O' = O \cup \{\mathsf{valid}_Y \,|\, Y \in O\} \cup \{\mathsf{stop}_X \,|\, X \in I\}$.

The triples $\langle X, \mathsf{valid}_X, \mathsf{stop}_X \rangle$ (for $X \in I$) and $\langle Y, \mathsf{valid}_Y, \mathsf{stop}_Y \rangle$ (for $Y \in O$) are to be thought of as *elastic input and output channels* of the system.

Let $\mathsf{transfer}_Z$ be a shorthand for $\mathsf{valid}_Z \wedge \neg \mathsf{stop}_Z$ and say that *transfer along Z occurs in a state s* if $s \models \mathsf{transfer}_Z$. Given a behavior $\sigma = (\sigma[0], \sigma[1], \sigma[2], \ldots)$ of an $[I, O]$-system and $Z \in I \cup O$, let σ_Z be the sequence (perhaps finite!) obtained from $\sigma.Z = (\sigma[0].Z, \sigma[1].Z, \sigma[2].Z, \ldots)$ by deleting all entries $\sigma[i].Z$ such that transfer along Z does not occur in $\sigma[i]$. The *transfer behavior* $\sigma^\intercal$ associated with σ is then defined by $\sigma^\intercal.Z = \sigma_Z$. If all sequences σ_Z are infinite, then $\sigma^\intercal$ is an $(I \cup O)$-behavior; in general, however, we only have $\sigma_Z \in \mathsf{type}(Z)^\omega$.

For each wire Z of an $[I, O]$-system $\mathcal{S}$ we introduce an auxiliary *transfer counter variable* tct_Z of type $\mathbb{Z}$. The counters serve for expressing system properties related to transfer. By definition, tct_Z is equal to the number of states that precede the current state and in which transfer along Z has occurred. That is, for every behavior σ of $\mathcal{S}$, we have $\sigma.\mathsf{tct}_Z = (t_0, t_1, \ldots)$, where t_k is the number of indices i such that $i < k$ and transfer along Z occurs in $\sigma[i]$. Note that the sequence $\sigma.\mathsf{tct}_Z$ is non-decreasing and begins with $t_0 = 0$.

The notation $\mathsf{min\_tct}_S$, for any subset S of $I \cup O$ will be used to denote the smallest of the numbers tct_Z, where $Z \in S$.

B. Definition of Elasticity

An elastic component, when ready to communicate over an output channel must remain ready until the transfer takes place.

Definition 6: The *persistence conditions* for an $[I, O]$-system $\mathcal{S}$ are given by

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{valid}_Y \wedge \mathsf{stop}_Y \Rightarrow (\mathsf{valid}_Y)^+ \wedge Y^+ = Y) \qquad (1)$$

for every $Y \in O$.

The conjunct $Y^+ = Y$ can be removed from (1) without affecting the definition of elastic machines (it follows from other conditions). The most useful consequence of persistence is the "handshake lemma":

$$\mathcal{S} \models \mathsf{G}\,\mathsf{F}\,\mathsf{valid}_Y \wedge \mathsf{G}\,\mathsf{F}\,\neg\mathsf{stop}_Y \;\Rightarrow\; \mathsf{G}\,\mathsf{F}\,\mathsf{transfer}_Y$$

Liveness of an elastic component is expressed in terms of token count: if all input channels have seen k transfers and there is an output channel that has seen less, then the communication on output channels with the minimum amount of transfer must be eventually offered. The following definition formalizes this,

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

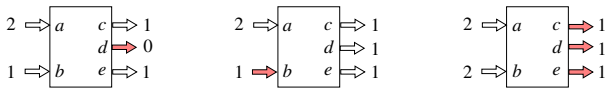


Fig. 5. Liveness: Only the hungriest channels (shaded) are being served. The numbers indicate the current token count at each channel.

together with a similar commitment to eventual readiness on input channels. (See also Figure 5.)

Definition 7: The *liveness conditions* for an $[I, O]$-system are given by

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_O = \mathsf{tct}_Y \wedge \mathsf{min\_tct}_I > \mathsf{tct}_Y \Rightarrow \mathsf{F}\,\mathsf{valid}_Y) \quad (2)$$

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_{I\cup O} = \mathsf{tct}_X \Rightarrow \mathsf{F}\,\neg\mathsf{stop}_X) \quad (3)$$

for every $Y \in O$ and every $X \in I$.

In practice, elastic components will satisfy simpler (but stronger) liveness properties; e.g. remove $\mathsf{min\_tct}_O \geq \mathsf{tct}_Y$ from (2) and replace $\mathsf{min\_tct}_{I\cup O} \geq \mathsf{tct}_X$ with $\mathsf{min\_tct}_O \geq \mathsf{tct}_X$ in (3). However, a composition of such components, while satisfying (2) and (3), may not satify the stronger versions of these conditions.

Consider single-channel $[I, O]$-systems satisfying the persistence and liveness conditions: an *elastic consumer* is a $[\{Z\}, \emptyset]$-system $\mathcal{C}$ satisfying (4) below; similarly, an *elastic producer* is a $[\emptyset, \{Z\}]$-system $\mathcal{P}$ satisfying (5) and (6).

$$\mathcal{C} \models \mathsf{G}\,\mathsf{F}\,\neg\mathsf{stop}_Z \quad (4)$$

$$\mathcal{P} \models \mathsf{G}\,(\mathsf{valid}_Z \wedge \mathsf{stop}_Z \Rightarrow (\mathsf{valid}_Z)^+) \quad (5)$$

$$\mathcal{P} \models \mathsf{G}\,\mathsf{F}\,\mathsf{valid}_Z \quad (6)$$

Let $\mathcal{C}_Z$ be the $\{Z, \mathsf{valid}_Z, \mathsf{stop}_Z\}$-system characterized by condition (4)—the largest (in the sense of behavior inclusion) of the systems satisfying this condition. Similarly, let $\mathcal{P}_Z$ be the $\{Z, \mathsf{valid}_Z, \mathsf{stop}_Z\}$-system characterized by properties (5) and (6). Finally, define *the* $[I, O]$*-elastic environment* to be the system

$$\mathsf{Env}_{I,O} = \bigsqcup_{X\in I} \mathcal{P}_X \sqcup \bigsqcup_{Y\in O} \mathcal{C}_Y.$$

Note that $\mathsf{Env}_{I,O}$ is only a system; it is not functional and so is not a machine.

When a system satisfying the persistence and liveness conditions (1-3) is coupled with a matching elastic environment, the transfer on all data wires never comes to a stall:

Lemma 3 (Liveness): If $\mathcal{S}$ satisfies (1-3), then for every behavior ω of $\mathcal{S} \sqcup \mathsf{Env}_{I,O}$, all the component sequences of the transfer behavior ω^T are infinite.

As an immediate consequence of Liveness Lemma, if $\mathcal{S}$ satisfies (1-3), then

$$\mathcal{S}^\mathsf{T} = \{\omega^\mathsf{T} \mid \omega \in \mathcal{S} \sqcup \mathsf{Env}_{I,O}\}$$

is a well-defined (I, O)-system.

Definition 8: An $[I, O]$-system $\mathcal{S}$ is an $[I, O]$*-elastic machine* if it satisfies the properties (1-3) and the associated system $\mathcal{S}^\mathsf{T}$ is deterministic.

The liveness conditions (2,3) are visibly related to causality at the transfer level: k transfers on the input channels imply k transfers on the output channels in the cooperating environment. Thus, it is not surprising (even though the proof is not obvious) that the determinism postulated in Definition 8 suffices to derive the causality of $\mathcal{S}^\mathsf{T}$:

Theorem 2: If $\mathcal{S}$ is an $[I, O]$-elastic machine, then $\mathcal{S}^\mathsf{T}$ is an (I, O)-machine.

In the situation of Definition 8, we say that $\mathcal{S}$ is an *elasticization* of $\mathcal{S}^\mathsf{T}$ and that $\mathcal{S}^\mathsf{T}$ is the *transfer machine* of $\mathcal{S}$.

IV. Elastic Networks

An *elastic network* $\mathcal{N}$ is given by a set of elastic machines $\mathcal{S}_1, \ldots, \mathcal{S}_m$ with no shared wires, together with a set of channel pairs $(X_1, Y_1), \ldots, (X_n, Y_n)$, where the X_i are n distinct input channels and the Y_i are n distinct output channels. As a network of standard machines, the elastic network $\mathcal{N}$ is defined by

$$\mathcal{N} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_m \mid X_i = Y_i, \mathsf{valid}_{X_i} = \mathsf{valid}_{Y_i}, \mathsf{stop}_{X_i} = \mathsf{stop}_{Y_i}\ (1 \leq i \leq n)\rangle$$

for which we will use the shorter notation

$$\mathcal{N} = \langle\!\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \parallel X_1 = Y_1, \ldots, X_n = Y_n \rangle\!\rangle.$$

We will define a graph that encodes the sequentiality information about the network $\mathcal{N}$ and prove in Theorem 4 that acyclicity of that graph implies that $\mathcal{N}$ is an elastic machine and that $\mathcal{N}^\mathsf{T} = \langle \mathcal{S}_1^\mathsf{T}, \ldots, \mathcal{S}_m^\mathsf{T} \mid X_1 = Y_1, \ldots, X_n = Y_n \rangle$.

A. Elastic Feedback

Elastic feedback is a simple case of elastic network:

$$\langle\!\langle \mathcal{S} \parallel P = Q \rangle\!\rangle = \langle \mathcal{S} \mid P = Q, \mathsf{valid}_P = \mathsf{valid}_Q, \mathsf{stop}_P = \mathsf{stop}_Q \rangle.$$

Definition 9: Suppose $\mathcal{S}$ is an elastic machine. An input-output channel pair (P, Q) will be called *sequential* for $\mathcal{S}$ if

$$\mathcal{S} \models \mathsf{G}\left(\begin{array}{c} \mathsf{min\_tct}_{I\cup O} = \mathsf{tct}_Q \\ \wedge\ \mathsf{min\_tct}_{I-\{P\}} > \mathsf{tct}_Q \end{array} \Rightarrow \mathsf{F}\,\mathsf{valid}_Q \right). \quad (7)$$

Condition (7) is a strengthening of the liveness condition (2) for channel Q. It expresses a degree of independence of the output channel Q from the input channel P; e.g., the first token at Q need not wait for the arrival of the first token at P. This independence can be achieved in the system by storing some tokens inside, between these two channels. Note that (7) does not guarantee that connecting channels P and Q would not introduce ordinary combinational cycles. Therefore the acyclicity condition in the following theorem is required to ensure (by Theorem 1) that the elastic feedback, viewed as an ordinary network, is a machine.

Theorem 3: Let $\mathcal{S}$ be an elastic machine and $\mathcal{F}$ the elastic feedback system $\langle\!\langle \mathcal{S} \parallel P = Q \rangle\!\rangle$. If the channel pair (P, Q) is sequential for $\mathcal{S}$, then: (a) the wire pair (P, Q) is sequential for $\mathcal{S}^\mathsf{T}$. If, in addition, $\Delta(\mathcal{F})$ is acyclic, then: (b) $\mathcal{F}$ is an elastic machine, and (c) $\mathcal{F}^\mathsf{T} = \langle \mathcal{S}^\mathsf{T} \mid P = Q \rangle$.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

B. Main Theorems

Sequentiality of two channel pairs $(P,Q),(P',Q)$ of an elastic machine does not imply their "simultaneous sequentiality"

$$\mathcal{S} \models \mathsf{G}\left(\begin{array}{c}\mathsf{min\_tct}_{I\cup O} = \mathsf{tct}_Q \\ \wedge\ \mathsf{min\_tct}_{I-\{P,P'\}} > \mathsf{tct}_Q\end{array} \Rightarrow \mathsf{F}\,\mathsf{valid}_Q\right).$$

This deviates from the situation with ordinary machines, where the analogous property holds and is instrumental in the proof of Combinational Loop Theorem.

To justify multiple feedback on elastic machines, we have thus to postulate that simultaneous sequentiality is true where required. Specifically, we demand that elastic machines come with simultaneous sequentiality information: If $\mathcal{S}$ is an $[I,O]$-elastic machine, then for every $Y \in O$ a set $\delta(Y) \subseteq I$ is given so that

$$\mathcal{S} \models \mathsf{G}\left(\begin{array}{c}\mathsf{min\_tct}_{I\cup O} = \mathsf{tct}_Y \\ \wedge\ \mathsf{min\_tct}_{I-\delta(Y)} > \mathsf{tct}_Y\end{array} \Rightarrow \mathsf{F}\,\mathsf{valid}_Y\right). \quad (8)$$

Note that if $P \in \delta(Q)$, then the pair (P,Q) is sequential, but the converse is not implied. A function $\delta\colon O \to 2^I$ with the property (8) will be called a *sequentiality interface* for $\mathcal{S}$.

For an $[I,O]$-elastic machine $\mathcal{S}$ with a sequentiality interface δ, we define $\Delta^{\mathrm{e}}(\mathcal{S},\delta)$ to be the graph with the vertex set $I \cup O$ and directed edges (X,Y) where $X \notin \delta(Y)$. By Theorem 3(a), $\Delta^{\mathrm{e}}(\mathcal{S},\delta)$ contains $\Delta(\mathcal{S}^{\mathsf{T}})$ as a subgraph.

Given an elastic network $\mathcal{N} = \langle\!\langle \mathcal{S}_1,\ldots,\mathcal{S}_m \,\|\, X_1 = Y_1,\ldots,X_n = Y_n\rangle\!\rangle$, where each $\mathcal{S}_i$ comes equipped with a sequentiality interface δ_i, its graph $\Delta^{\mathrm{e}}(\mathcal{N})$ is by definition the direct sum of graphs $\Delta^{\mathrm{e}}(\mathcal{S}_1,\delta_1),\ldots,\Delta^{\mathrm{e}}(\mathcal{S}_m,\delta_m)$ with each vertex X_i $(1 \le i \le n)$ identified with the corresponding vertex Y_i.

Theorem 4: If the graphs $\Delta(\mathcal{N})$ and $\Delta^{\mathrm{e}}(\mathcal{N})$ are acyclic, then the network system $\mathcal{N}$ is an elastic machine, the corresponding non-elastic system $\bar{\mathcal{N}} = \langle \mathcal{S}_1^{\mathsf{T}},\ldots,\mathcal{S}_m^{\mathsf{T}} \,|\, X_1 = Y_1,\ldots,X_n = Y_n\rangle$ is a machine, and $\mathcal{N}^{\mathsf{T}} = \bar{\mathcal{N}}$.

As in Theorem 3, acyclicity of $\Delta(\mathcal{N})$ is needed to ensure (by Theorem 1) that $\mathcal{N}$ defines a machine. Elasticization procedures (e.g. [4]) will typically produce elastic components with enough sequential input-output wire pairs, so that $\Delta(\mathcal{N})$ will be acyclic as soon as $\Delta^{\mathrm{e}}(\mathcal{N})$ is acyclic.

Note, however, that cycles in $\Delta^{\mathrm{e}}(\mathcal{N})$ need not correspond to combinational cycles in $\mathcal{N}$ seen as an ordinary network, since empty buffers with sequential elements cutting the combinational feedbacks may be inserted into $\mathcal{N}$. Even though non-combinational in the ordinary sense, these cycles contain no tokens and therefore no progress along them can be made.

Theorem 4 impies that insertion of empty elastic buffers does not affect the basic functionality of an elastic network, as illustrated in Figure 2(b).

Definition 10: An *empty elastic buffer* is an elastic machine $\mathcal{S}$ such that $\mathcal{S}^{\mathsf{T}} = \mathsf{Conn}(X,Y)$ for some X,Y.

Theorem 5 (Buffer Insertion Theorem): Suppose $\mathcal{B}$ is an empty elastic buffer with channels X,Y. Let $\mathcal{N} = \langle\!\langle \mathcal{S}_1,\ldots,\mathcal{S}_m \,\|\, X_1 = Y_1,\ldots,X_n = Y_n\rangle\!\rangle$ and $\mathcal{M} = \langle\!\langle \mathcal{B},\mathcal{S}_1,\ldots,\mathcal{S}_m \,\|\, X = Y_1, X_1 = Y, X_2 = Y_2,\ldots,X_n = Y_n\rangle\!\rangle$. If $\Delta(\mathcal{N})$, $\Delta(\mathcal{M})$, and $\Delta^{\mathrm{e}}(\mathcal{N})$ are acyclic, then $\mathcal{M}$ is an elastic machine, and $\mathcal{M}^{\mathsf{T}} = \mathcal{N}^{\mathsf{T}}$.

The precise relationship between graphs $\Delta(\mathcal{M})$ and $\Delta(\mathcal{N})$ can be easily described. In practice they are at the same time acyclic or not, as a consequence of sequentiality of sufficiently many input-output wire pairs of $\mathcal{B}$.

V. Elastic vs. Patient Systems

Elastic machines and *patient processes* of [2] provide two formalizations of the intuitive concept of latency-insensitive circuits. In this section we address their connections and differences. We begin with an overview of [2], using a minimalistic approach and terminology that differs from the original. We believe, however, that Definition 11 below matches the original definion accurately in most important aspects.

A. Patient Systems

The notation A^* is for the set of finite sequences over A. A *finitary* W-system, by definition, is a set of behaviors σ such that $\sigma.w$ is a finite sequence for every $w \in W$.

A *stalling stream* over A is a stream over $A \cup \{\Box\}$. We will refer to $\Box$ as the *bubble* and to elements of A as *tokens*. We will consider only stalling streams that contain finitely many tokens. If a is such a stream, let $\overline{a} \in A^*$ denote the sequence over A obtained by dropping all bubbles from a. Clearly, a is determined by $\overline{a}$ and the sequence $\partial(a) \in \mathbb{N}^*$ of lengths of bubble sequences between consecutive tokens of a. For example, if

$$a = (\Box,\Box,7,\Box,4,5,\Box,\Box,\Box,8,\ldots) \quad (9)$$

we have $\overline{a} = (7,4,5,8,\ldots)$ and $\partial(a) = (2,1,0,3,\ldots)$. Two stalling streams a,b are *latency equivalent*, written $a \mathrel{\mathring{=}} b$, when $\overline{a} = \overline{b}$. Note that $a \mathrel{\mathring{=}} \overline{a}$.

By definition, a *stalling W-system* is a set of behaviors σ such that for every $w \in W$, $\sigma.w$ is a stalling stream over $\mathsf{type}(w)$. Latency equivalence extends to W-behaviors and W-systems: $\sigma \mathrel{\mathring{=}} \tau$ iff $\sigma.w \mathrel{\mathring{=}} \tau.w$ holds for every $w \in W$; $\mathcal{S} \mathrel{\mathring{=}} \mathcal{S}'$ iff for every $\sigma \in \mathcal{S}$ $(\sigma \in \mathcal{S}')$ there exists $\tau \in \mathcal{S}'$ $(\tau \in \mathcal{S})$ such that $\sigma \mathrel{\mathring{=}} \tau$.

A stalling W-system $\mathcal{S}$ determines a standard finitary W-system $\mathcal{S}^{\mathsf{T}} = \{\overline{\sigma} \mid \sigma \in \mathcal{S}\}$, where $\overline{\sigma}$ is given by $\overline{\sigma}.w = \overline{\sigma.w}$ (for all $w \in W$). Clearly, $\mathcal{S}^{\mathsf{T}} \mathrel{\mathring{=}} \mathcal{S}$.

Stalling the k-th token of a by d steps produces a latency equivalent stream that will be denoted $\mathsf{stall}(a,k,d)$. Omitting the easy definition, we give an example: if a is as in (9), then

$$\mathsf{stall}(a,1,3) = (\Box,\Box,7,\Box,\Box,\Box,\Box,4,5,\Box,\Box,\Box,8,\ldots)$$

Definition 11: Let $\prec$ be a well-founded order[1] on W and let $D > 0$. A *patient W-system* (relative to $\prec$ and D) is a

[1] Introduction of a well-founded ordering of wires is motivated in [2] with the purpose of modeling combinational dependencies, but such dependencies in patient systems are not discussed in any detail. Moreover, the ordering of wires is implicitly assumed to be *total* in [2], which is somewhat unnatural. For instance, when constructing a patient adder with inputs u,v and output w, one has two ordering choices: $u \prec_1 v \prec_1 w$ and $v \prec_2 u \prec_2 w$. It is not clear that a patient adder in the $\prec_1$-sense will be patient in the $\prec_2$-sense too.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

stalling system $\mathcal{P}$ such that for every $\sigma \in \mathcal{P}$, every $u \in W$, and every $k \geq 0$ there exists $\sigma' \in \mathcal{P}$ such that

(Pat-1) $\quad \sigma'.u = \mathsf{stall}(\sigma.u, k, 1)$

and for every $v \neq u$ there exists $d_v \leq D$ such that

$$\text{(Pat-2)} \quad \sigma'.v = \begin{cases} \mathsf{stall}(\sigma.v, k, d_v) & \text{if } u \prec v \\ \mathsf{stall}(\sigma.v, k+1, d_v) & \text{otherwise} \end{cases}$$

The main results of [2] can now be summarized:

1) a theorem saying that the composition of patient systems (with the same W, $\prec$, and D) is a patient system;
2) the definition and analysis of *patient buffers*, i.e. patient systems $\mathcal{B}$ such that $\mathcal{B}^{\mathsf{T}} = \mathsf{Conn}^{\mathrm{fin}}(u, v)$—the finitary connection system;
3) a general construction that, for a given finitary system $\mathcal{M}$ without combinational dependencies (model of a Moore machine), produces a patient system $\mathcal{P}$ such that $\mathcal{P} \triangleq \mathcal{M}$.

B. Comparison

The formalization given by patient systems is at a higher level of abstraction. While elastic machines deal explicitly with handshaking signals between communicating systems, patient systems communicate purely in the token/bubble language.

Given an elastic (as defined in Section III) $[I, O]$-system $\mathcal{E}$, the corresponding stalling $(I \cup O)$-system $\mathcal{E}^{\Box}$ is obtained by projecting the finite-transfer behaviors of $\mathcal{E}$ to data wires and replacing data items on each wire with $\Box$ at all cycles where transfer along that wire does not occur. Precisely, let $\mathcal{E}^{\mathsf{F}}$ be the subset of $\mathcal{E}$ consisting of all behaviors ω such that $\omega^{\mathsf{T}}.Z$ is finite for all channels Z.[2] Then, given $\omega \in \mathcal{E}^{\mathsf{F}}$, we define a stalling $(I \cup O)$-behavior $\omega^{\Box}$ by

$$(\omega^{\Box}.Z)[i] = \begin{cases} (\omega.Z)[i] & \text{if } (\omega.\mathsf{valid}_Z)[i] \wedge \neg(\omega.\mathsf{stop}_Z)[i] \\ \Box & \text{otherwise} \end{cases}$$

and finally we define the stalling system $\mathcal{E}^{\Box}$ as the set of all such behaviors $\omega^{\Box}$. Clearly, the system $(\mathcal{E}^{\Box})^{\mathsf{T}}$ is the finitary version of the standard machine $\mathcal{E}^{\mathsf{T}}$.

Now we can address some questions pertinent to the comparison of patient processes vs. elastic machines.

Are patient processes more general? The answer is "no" because there exist elastic machines $\mathcal{E}$ such that $\mathcal{E}^{\Box}$ is not patient. To see this, consider an elastic machine $\mathcal{E}$ that starts offering new valid outputs on channel u only on even cycles. (The existence of such elastic machines is obvious.) Observe that $\sigma.u = (\Box, 7, 9, \ldots)$ is possible for some behavior σ of $\mathcal{E}^{\Box}$ (token 7, even though transmitted on cycle 1 was first offered on cycle 0). Then $\mathsf{stall}(\sigma.u, 0, 1) = (\Box, \Box, 7, 9, \ldots)$ must also be part of a behavior of $\mathcal{E}^{\Box}$, by condition (Pat-1) of Definition 11. This implies that token 9 is first offered on cycle 3, contrary to our assumption.

The above example can be viewed as an indication that the condition (Pat-1) is too restrictive. It would be interesting to see if an appropriate modification of (Pat-1) results in a definition of patient processes that captures elastic machines.

[2]One can prove that $\mathcal{E}$ is the set of all limits of behaviors of $\mathcal{E}^{\mathsf{F}}$ and so $\mathcal{E}$ is determined by $\mathcal{E}^{\mathsf{F}}$.

Are elastic machines more general? The answer is an easy "no" since, for example, the set of all possible stalling W-behaviors is a patient system in the sense of Definition 11. However, if one adds to Definition 11 a reasonable requirement that a patient system be a machine, the answer is not immediately clear.

Which formalization is easier to use? Without offering a definitive answer, we would argue that verifying that a low-level design (RTL, say) implements an elastic machine would be easier than verifying that it implements a patient system. The bottom line is that the conditions for a system to be an elastic machine are expressible as temporal properties of suitably constructed infinite-state models. This is not obvious for the determinism condition for $\mathcal{S}^{\mathsf{T}}$ in Definition 8, but can be done by replacing determinism with causality and introducing auxiliary variables for sequences of transferred values over channels. Even though (e.g., because of infinite counters involved) these conditions are not directly checkable by the existing model checking technology, there are palpable opportunities to find manageable stronger conditions that taken together imply elasticity (e.g., postulating a limit on the token count differences between channels eliminates the need for infinite counters). On the other hand, the definition of a patient system, being of the form "for every behavior σ, there exists a behavior σ' such that ..." appears to us to be intrinsically more complex. Our only positive conclusion, however, is that the mechanical checking of either of the definitions is an open problem deserving further study.

VI. Conclusion

We have presented a theory of elastic machines that gives an easy-to-check condition for the compositional theorem of the form "an elasticization of a network of ordinary components is equivalent to the network of components' elasticizations". Verification of a particular implementation is reduced to proving that conditions of Definition 8 are satisfied for all elastic components used, and that the graph $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}})$ is acyclic for every network $\mathcal{N}$ to which the elasticization is applied. While the definition of the graphs Δ^{e} may appear complex because of the sequentiality interfaces involved, it should be noted that the elasticization procedures, e.g. [4], are reasonably expected to completely preserve sequentiality: a channel P belongs to $\delta(Q)$ if the wire-pair (P, Q) is sequential in the original non-elastic machine. This ensures $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}}) = \Delta(\mathcal{N})$ and so testing for sequentiality is done at the level of ordinary networks.

Future work will be focused on proving correctness of particular elasticization methods, on techniques for mechanical verification of elasticity, and on extending the theory to more advanced protocols.

Acknowledgments: Luca Carloni clarified some details of [2]. Ken McMillan pointed out several inaccuracies in a previous version of the paper and further clarified [2] for us. Gerard Berry, Ching-Tsun Chou, John Harrison, and the anonymous reviewers provided useful remarks. We are grateful for all the help we received.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

REFERENCES

[1] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft book, available at http://www.esterel.org, version 3, July 1999.

[2] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 20(9):1059–1076, September 2001.

[3] L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro, Special Issue on Systems on Chip*, 22(5):12, October 2002.

[4] J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proc. Digital Automation Conference (DAC)*, July 2006.

[5] S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.

[6] H. M. Jacobson et al. Synchronous interlocked pipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, 2002.

[7] S. Krstić, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. Available at `www.lsi.upc.edu/~jordicf/gavina/BIB/reports/fmcad06_ext.pdf`, 2006.

[8] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[9] E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. Invited paper in *Foundations of Interface Technologies (FIT 2005)*, available at `http://ptolemy.eecs.berkeley.edu/publications`.

[10] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Proc. 4th Int. Conf. on the Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 272–285, 1998.

[11] J. Matthews. Recursive function definition over coinductive types. In *TPHOLs '99: Proc. the 12th Int. Conf. on Theorem Proving in Higher Order Logics*, pages 73–90, London, UK, 1999. Springer-Verlag.

[12] S. Suhaib, D. Berner, D. Mathaikutty, J.-P. Talpin, and S. Shukla. Presentation and formal verification of a family of protocols for latency insensitive design. Technical Report 2005-02, FERMAT, Virginia Tech, 2005.

[13] J. Vuillemin. On circuits and numbers. *IEEE Transactions on Computers*, 43(8):868–879, 1994.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

Finite Instantiations for Integer Difference Logic*

Hyondeuk Kim Fabio Somenzi
Department of Electrical and Computer Engineering
University of Colorado at Boulder, CO 80309-0425
{Hyondeuk.Kim,fabio}@Colorado.EDU

Abstract

The last few years have seen the advent of a new breed of decision procedures for various fragments of first-order logic based on propositional abstraction. A lazy satisfiability checker for a given fragment of first-order logic invokes a theory-specific decision procedure (a *theory solver*) on (partial) satisfying assignments for the abstraction. If the assignment is found to be consistent in the given theory, then a model for the original formula has been found. Otherwise, a refinement of the propositional abstraction is extracted from the proof of inconsistency and the search is resumed. We describe a theory solver for *integer difference logic* that is effective when the formula to be decided contains equality and disequality (negated equality) constraints so that the decision problem partakes of the nature of the pigeonhole problem. We propose a reduction of the problem to propositional satisfiability by computing bounds on a sufficient subset of solutions, and present experimental evidence for the efficiency of this approach.

1. Introduction

Decision procedures for fragments of first-order logic have been the subject of intense scrutiny in the last few years. On the one hand, emerging applications like model checking of infinite state systems rely on such decision procedures for tasks like predicate abstraction [1]. On the other hand, algorithmic advances have significantly increased the range of problems that can be tackled, and hence have stimulated interest.

In particular, dramatic increase in performance of propositional satisfiability (SAT) solvers has led to the development of decision procedures that rely on the *propositional abstraction* of formulae from more expressive logics like the logic of linear arithmetic constraints, Presburger arithmetic, or the logic of equality and uninterpreted function symbols (EUF). The propositional abstraction of a formula is obtained by replacing the atomic formulae of the specific theory (e.g., $x-y \leq 5$ or $f(x) = f(y)$, where f is an uninterpreted function symbol) with fresh propositional variables. The satisfying assignments of the abstraction map to conjunctions of literals in the original formula that can be checked for consistency with theory-specific procedures. If such a procedure establishes consistency, then the given formula is satisfiable and the enumeration terminates. Otherwise, from the proof of inconsistency a refinement of the propositional abstraction is extracted and the search is resumed.

There are several ways to combine the propositional reasoning engine with the theory-specific procedures. One broad classification is the one into *lazy* and *eager* approaches. A lazy solver produces an initial propositional approximation that is concise and possibly quite coarse; it relies on the refinements during the enumeration of solutions. By contrast, an eager solver adds constraints to the initial propositional abstraction that embody known relationships among the literals. An example is given by the constraints that encode transitivity of equality. The most effective solvers often adopt elements of both approaches and tailor their strategies to the theory (theories) at hand.

In this paper we focus on *Integer Difference Logic* (IDL), in which arithmetic atomic formulae constrain the difference between the values of pairs of integer variables. This logic finds extensive application to problems involving timing and scheduling constraints, resource allocation, and program analysis. IDL is closely related to *Real Difference Logic* (RDL), to the point that a decision procedure for the latter based on propositional abstraction also works for the former, as long as the coefficients are integers. It is sufficient to rewrite *equality* constraints (of the form $x - y = n$) as the conjunction of two inequalities. However, if an equality constraint is negated, then the conjunction turns into a disjunction, which requires case splitting in the enumeration of the propositional solutions. In contrast, we propose an approach that does not decompose equalities and their negations; rather, it converts the problem of checking satisfiability of a conjunction of arithmetic atomic formulae into a set of propositional satisfiability checks—whose cardinality is bounded by the number of maximal strongly connected components (SCC) of a suitable constraint graph.

The conversion to propositional satisfiability that we propose is based on the ability to bound the values of the integer variables that appear in the formula. While in general such bounds do not exist, we show that to decide satisfiability of a set of constraints whose graph is a single SCC it is sufficient to consider a subset of the solutions for which bounds are easily established. We also show how the general case can be efficiently solved given solutions for the individual SCCs of the constraint graph. Experimental study shows that our new approach greatly improves the efficiency of our decision procedure for problem instances in which disequalities play a significant role, and makes it very competitive with respect to state-of-the-art tools.

The rest of this paper is organized as follows: Section 2 reviews background and introduces notation. Section 3 discusses the bounds on solutions, while Sect. 4 delves into the details of our theory solver. After a brief survey of related work in Sect. 5, experiments are presented in Sect. 6, and conclusions are offered in Sect. 7.

2. Preliminaries

Let P be a set of propositional variables and X a set of integer-valued variables. We define inductively *integer difference logic* (IDL) formulae as follows.

- $p \in P$ is a (propositional atomic) IDL formula.
- $x - y \leq n$ and $x - y = n$ are (arithmetic atomic) IDL formulae, for $x, y \in X$, $n \in \mathbb{Z}$.
- If φ and ψ are IDL formulae, so are $\varphi \wedge \psi$ and $\neg\varphi$.

*This work was supported by SRC contract 2005-TJ-920.

The following abbreviations are also defined:

$$x - y < n \doteq x - y \leq n - 1 \quad x - y \neq n \doteq \neg(x - y = n)$$
$$x = y \doteq (x - y = 0) \quad x \neq y \doteq \neg(x = y)$$
$$\varphi \vee \psi \doteq \neg(\neg\varphi \wedge \neg\psi) \ .$$

A literal is an atomic formula, or the negation of an atomic formula. A *clause* is the disjunction of a set of literals, and a formula in *conjunctive normal form* (CNF) is the conjunction of a set of clauses. Note that $x - y = n$ could be defined as an abbreviation $((x - y \leq n) \wedge (x - y \geq n))$. We choose not to do so to stress that our algorithm does not split equalities or disequalities (as mentioned in Sect. 1) and also to keep the definition of clausal formulae simple.

A *model* for an IDL formula φ is a pair of functions $\alpha : X \to \mathbb{Z}$ and $\beta : P \to \{\mathsf{false}, \mathsf{true}\}$ such that replacing each variable $x \in X$ with $\alpha(x)$, and every variable $p \in P$ with $\beta(p)$ in φ produces a true statement. A formula is *satisfiable* if it has a model, and is *valid* if every assignment is a model. If a conjunction of literals φ is not satisfiable, then a (minimal) *explanation* for the unsatisfiability of φ is the conjunction of a (minimal) subset of the literals in ψ which is not satisfiable.

Propositional logic is the fragment of IDL obtained by omitting the rule that defines arithmetic atomic formulae. Efficient algorithms to decide the satisfiability of propositional logic formulae are based on the DPLL procedure [7, 6], and exploit techniques like clause recording, conflict analysis, nonchronological backtracking, and fast Boolean constraint propagation [23, 20].

In recent times, decision procedures for IDL, and other fragments of quantifier-free first-order logic, have been based on the DPLL procedure as well. Given a set of propositional variables B such that $B \cap P = \emptyset$, one obtains a propositional formula φ^b from an IDL formula φ by replacing each arithmetic atomic subformula of φ with a distinct variable from B. The resulting formula φ^b is unsatisfiable only if φ is unsatisfiable. Each model of φ^b corresponds to a conjunction of literals of φ. Given a decision procedure for the conjunction of arithmetic atomic propositions in IDL (a *theory solver*), one therefore derives a complete decision procedure for IDL by enumerating the models of φ^b, extracting from each of them the corresponding conjunction of arithmetic atomic propositions and their negations, and checking these conjunctions for satisfiability using the theory solver. In the following, we refer to the conjunction of a set of arithmetic literals as a *set of IDL constraints*.

An edge integer-labeled directed graph is a triple $G = (V, E, \lambda)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $\lambda : E \to \mathbb{Z}$ is an edge labeling function. A *strongly connected component* (SCC) of G is a subgraph G' of G such that every two nodes of G' are connected by a path in G'. SCC G' is *maximal* if no subgraph of G that is a proper superset of G' is an SCC; it is *trivial* if it consists of one vertex and no arcs. The maximal SCCs of G define a partition of V. The *SCC quotient graph* $\widehat{G} = (\widehat{V}, \widehat{E})$ of G is a directed acyclic graph with one vertex for each maximal SCC of G and an edge $(A, B) \in \widehat{E}$ if and only if there exist $x \in A$ and $y \in B$ such that $(x, y) \in E$.

Given a distinguished source vertex $s \in V$, distances of all vertices from s are well defined provided there exists no *negative cycle* in G; that is, no cycle such that the sum of the labels on the edges along the cycle is negative. The Bellman-Ford algorithm [5] reports negative cycles if they are present, and computes the distance $\delta(x)$ of each vertex in V from the source s otherwise. The *slack* of an edge $(x, y) \in E$ is given by $\sigma((x, y)) = \lambda((x, y)) - (\delta(y) - \delta(x))$. It is easy to see that for all $e \in E$, $\sigma(e) \geq 0$ and that $\sigma((x, y)) = 0$ if and only if (x, y) is on a shortest path from s to y in G. Distances and slacks obviously depend on the choice of source vertex.

Given a (finite) set I of inequality constraints (i.e., of the form $x - y \leq n$), their *constraint graph* $G = (V, E, \lambda)$ is a labeled directed graph defined as follows:

- $V \subseteq X$ is the set of variables appearing in the constraints in I.
- There is an arc $(x, y) \in E$ with $\lambda((x, y)) = n$ if and only if there is a constraint $y - x \leq n$ in I.

It is well known [5] that I is satisfiable if and only if G contains no negative cycle. In fact, adding both sides of the constraints forming a cycle of length w, one gets $0 \leq w$, which is not satisfiable when $w < 0$. If, on the other hand, no negative cycle exists in G, then one can find a model for I by solving a single-source shortest-path problem on an augmented graph G_a, obtained from G by adding a new reference vertex x_r and arcs labeled 0 from x_r to all the other vertices. Let $\delta(x)$ be the distance of $x \in V$ from x_r in G_a. Then δ is a model for I. It is also well known that, given a model of I, $\alpha : V \to \mathbb{Z}$, and a constant, $c \in \mathbb{Z}$, the assignment $\alpha' : V \to \mathbb{Z}$ defined by $\alpha'(x) = \alpha(x) + c$ is also a model of I, because $\alpha'(x) - \alpha'(y) = \alpha(x) - \alpha(y)$. This observation allows an easy encoding of *range constraints* in IDL. A set of constraints $\{l_i \leq x_i \leq u_i\}$ is translated to $\{x_i - y \leq u_i\} \cup \{y - x_i \leq -l_i\}$, where y is a fresh variable. The solution α obtained from the constraint graph is then translated so that $\alpha'(y) = 0$.

Since integer labels imply integer distances, if the right-hand sides of the constraints are integer-valued, and the constraints are satisfiable when the variables are real-valued, then an integer-valued solution is also guaranteed to exist. Loosely speaking, the satisfiability problem for *inequalities* is the same for IDL and real difference logic (RDL). Adding *equality* constraints to the inequalities does not change this state of affairs: Given a constraint $x - y = n$, one replaces x by $y + n$; if no immediate inconsistencies arise, one continues with the construction of the constraint graph. In contrast, if disequality constraints (i.e., negations of equalities) are allowed, an unsatisfiable conjunction of IDL constraints may be satisfiable when regarded as an RDL formula. An example is given by $\bigwedge_{1 \leq i \leq p}(1 \leq x_i \leq h) \wedge \bigwedge_{1 \leq i < j \leq p}(x_i \neq x_j)$, which exemplifies the pigeonhole principle.[1]

3. Bounds on Solutions

It was recalled in Sect. 2 that from a solution α to a set of inequality constraints, one can derive a family of solutions $\{\alpha + n\}$. In general, however, not all solutions are obtained one from the other by *translation*. Consider the constraints $\{(x - y \leq 1), (y - x \leq 0)\}$. It is easy to verify that the two assignments $\alpha_1(x) = 0$, $\alpha_1(y) = 0$ and $\alpha_2(x) = 1$, $\alpha_2(y) = 0$ satisfy the constraints, though there is no n such that $\alpha_1 = \alpha_2 + n$. Such solutions are called *independent*. In general, there may be several families of independent solutions, and therefore, multiple distinct solutions that assign a given value to a distinguished variable. The following result characterizes these sets of solutions and forms the basis for our treatment of disequality constraints in IDL.

Theorem 1 *Let I be a set of inequality constraints. Let $G = (V, E, \lambda)$ be the constraint graph associated to I. Suppose that G contains no negative cycle and consists of one maximal SCC. For $x \in V$ and $n \in \mathbb{Z}$, let S_x^n be the set of solutions $\alpha : V \to \mathbb{Z}$*

[1] This does not contradict what was observed in Sect. 1 because $x \neq y$ translates into $(x < y) \vee (y < x)$ for RDL, but translates into $(x \leq y - 1) \vee (y \leq x - 1)$ for IDL.

to I such that $\alpha(x) = n$. Then, for each vertex $y \in V$, there exist bounds y_l and y_u such that for every solution in S^n_x, $y_l \leq \alpha(y) \leq y_u$:

$$\forall y \in V \,.\, \exists y_l, y_u \in \mathbb{Z} \,.\, \forall \alpha \in S^n_x \,.\, y_l \leq \alpha(y) \leq y_u \ .$$

PROOF. By definition of SCC, every vertex in V is reachable from x in G; likewise, x is reachable from any vertex in G. Let δ_{xy} be the distance of y from x (the length of a shortest path). Such a distance is defined because there are no negative cycles in G. Adding both sides of all the constraints along the path yields $y - x \leq \delta_{xy}$. Therefore, for every solution $\alpha \in S^n_x$, it must be $\alpha(y) \leq n + \delta_{xy}$. Said otherwise, $y_u = n + \delta_{xy}$. For the lower bound, let δ_{yx} be the distance of x from y in G. Then, for every solution $\alpha \in S^n_x$, it must be $\alpha(y) \geq n - \delta_{yx}$; that is, $y_l = n - \delta_{yx}$. □

Satisfaction of disequalities is not affected by translation. Therefore, a set of constraints including both inequalities and disequalities is satisfiable if and only if there exists a solution α such that $\alpha(x) = n$. This allows us to limit the search to the set S^n_x. Theorem 1 asserts that solutions in this set are bounded. Hence, we can resort to finite instantiations to find them. Specifically, we can encode each integer variable with binary variables and translate the satisfiability problem for a conjunction of inequality and disequality constraints into a propositional satisfiability problem.

Theorem 1 applies when the constraint graph consists of one maximal SCC. If that is not the case, we examine the SCC quotient graph one SCC at the time. If there is no negative cycle in the constraint graph G, the only reason for unsatisfiability is the inability to satisfy the disequalities within some SCC of G. Therefore, if the finite instantiation of each SCC is satisfiable, the entire set of constraints is satisfiable. This can be shown as follows.

Let G be the constraint graph. Extend G by adding one edge for every disequality constraint $x - y \neq n$ (where n may be 0) such that x and y belong to different SCCs. Let $\preceq$ be the preorder defined by $u \preceq v$ if there is a path in G from u to v. (The preorder is updated after each edge addition.) If $x \preceq y$, add $y - x \leq -n - 1$ to E; if $y \preceq x$, add $x - y \leq n - 1$. If x and y are not comparable in the preorder, add either $y - x \leq -n - 1$ or $x - y \leq n - 1$, but not both. Note that adding these edges does not create cycles, and therefore does not change the SCCs of G. (See Sect. 4.)

Let $\widehat{G} = (\widehat{V}, \widehat{E})$ be the SCC quotient graph of the extended G. Consider the vertices in $\widehat{V}$ starting from the minimal SCCs (those with no predecessors) and proceeding in a chosen topological order. Let A_i be the i-th SCC in that order and let α_i be a solution for the constraints corresponding to its edges. Inductively assume that β_{i-1} is a solution for the constraints in the subgraph induced by $\bigcup_{0<j<i} A_j$. Let k be the maximum amount by which any constraint corresponding to an edge into A_i is violated. (Let $k = 0$ if no such violation exists.) Finally, let $\alpha'_i = \alpha_i - k$. Then, $\beta_i = \beta_{i-1} \cup \alpha'$ is a solution for the constraints in the subgraph induced by $\bigcup_{0<j\leq i} A_i$.

4. Algorithm

We assume a decision procedure for IDL based on propositional abstraction. The given IDL formula φ is translated into a propositional formula φ^b as described in Sect. 2. A *propositional reasoning engine* enumerates the satisfying assignments to φ^b and calls the *theory solver* to determine whether those satisfying assignments correspond to consistent assignments to the integer-valued variables.

The theory solver for IDL is relatively efficient. Therefore, it is advantageous to call it also on partial assignments to terminate the fruitless search of part of the state space, or to learn so-called *theory consequences* [21]. Our implementation follows this approach, though the equality constraints are split and the full check for inconsistencies due to disequalities is applied only to complete assignments. (See lines 38–42 of Fig. 1.) We omit the details of the incremental implementation of the Bellman-Ford algorithm. The interested reader is referred to [26].

4.1 The Theory Solver

The theory solver is called with a collection of arithmetic literals whose corresponding propositional literals are true in a model of the propositional formula φ^b; it then decides whether there is an assignment to the integer-valued variables that satisfies the conjunction of all those literals. The first step is to obtain a set of arithmetic atomic formulae (without negations) from the given set of literals. The given literals are rewritten according to their form:

1. $x - y \leq n$: unchanged;
2. $x = y$: unchanged;
3. $x - y = n$, with $n \neq 0$: split into $(x - y \leq n) \wedge (y - x \leq -n)$;
4. $\neg(x - y \leq n)$: rewritten as $y - x \leq -n - 1$;
5. $\neg(x = y)$: rewritten as $x \neq y$;
6. $\neg(x - y = n)$, with $n \neq 0$: rewritten as $x - y \neq n$.

Constraints of type 1, 3, and 4 are *inequalities* (I). Constraints of type 2 are *equalities* (Q), and finally, constraints of type 5 and 6 are *disequalities* (D). Specifically, constraints of type 5 form the set $D_0 \subseteq D$. Let $C = I \cup Q \cup D$.

The theory solver, whose pseudocode is shown in Figures 1 and 2, adopts the *layered* approach of MathSAT [4]. For IDL, it considers three main layers: equalities, inequalities, and disequalities. Let $X_= \subseteq X$ be the set of integer-valued variables appearing in Q. The theory solver creates an undirected equality graph $\mathcal{Q} = (X_=, \Gamma)$, where

$$\Gamma = \{\{x_i, x_j\} : x_i = x_j \in Q\} \ .$$

The vertices of $\mathcal{Q}$ are in the same class if they are made equivalent by the equality constraints. The feasibility of Q with D_0 is checked by comparing the equivalence class of the two vertices of each disequality constraint in D_0. If two vertices are in the same class, an explanation of infeasibility is returned. If the set of equality constraints is feasible, the variables in the same class are merged into a single variable, and some simplified constraints in D_0 and I are dropped from the set.

The algorithm continues by checking the feasibility of the set of inequality constraints. Let $V \subseteq X$ be the set of integer-valued variables appearing in I. The theory solver creates a constraint graph $G = (V, E, \lambda)$ from I as explained in Sect. 2. The Bellman-Ford algorithm is run on G. If a negative cycle is found, the set I is infeasible; a negative cycle with a subset of Q provides the explanation of infeasibility. Equality constraints are involved in the explanation if the constraints on the negative cycle were obtained by simplification in the equality layer. If there is no negative cycle in G, the set $I \cup Q$ is feasible; therefore a solution $\delta : V \to \mathbb{Z}$ is returned by the Bellman-Ford algorithm.[2]

The (simplified) set I combined with D is considered in the next step. Let G_0 be the subgraph of G such that the edges with non-zero slacks for solution δ are removed from G. Since the slacks of

[2]The algorithm is, in principle, applied to the augmented graph G_a described in Sect. 2. In practice, no augmentation of G is required: it suffices to initialize all distances to 0.

```
1   TheorySolver (C) {
2       Q = CreateEqualityGraph (Q);
3       Explanation = CheckFeasibilityOfEqualityConstraints (Q, D_0);
4       if (Explanation ≠ SAT) return Explanation;
5       else {
6           MergeVariablesInSameClass (Q);
7           DropSimplifiedConstraints (C);
8           return CheckFeasibilityOfInequalityConstraints (I);
9       }
10  }

11  CheckFeasibilityOfInequalityConstraints (I) {
12      G = CreateConstraintGraph (I);
13      NegCycle = BellmaFordAlgorithm (G);
14      if (NegCycle) {
15          return GenerateExplanationFromNegCycle (NegCycle);
16      } else {
17          SCC = GenerateZeroSlackSccOfConstraintGraph (G);
18          Explanation = CheckFeasibilityOfZeroSlackScc (SCC, D);
19          if (Explanation ≠ SAT) return Explanation;
20          else {
21              SCC' = GeneratePositiveSlackSccOfConstraintGraph (G);
22              return CheckFeasibilityOfPositiveSlackScc (SCC', D);
23          }
24      }
25  }

26  CheckFeasibilityOfZeroSlackScc (SCC, D) {
27      For each d ∈ D {
28          Explanation = CheckFeasibilityOfDisequalityConstraint (SCC, d);
29          if (Explanation ≠ SAT) return Explanation;
30          else DropValidConstraint (d,D);
31      }
32      return SAT;
33  }

34  CheckFeasibilityOfPositiveSlackScc (SCC', D) {
35      for each scc' ∈ SCC' {
36          (L, U) = GenerateBoundsForEachVariableInScc (SCC');
37          Explanation = CheckFeasibilityOfBoundsWithClique(SCC', D, L, U);
38          if (Explanation = UNDECIDED or Explanation = PROB_SAT and assignment is complete) {
39              CNF = SmallDomainEncodingForConstraintsInScc (SCC', D, L, U);
40              Explanation = SatSolver (CNF);
41              if (Explanation ≠ SAT) return Explanation;
42          }
43          else return Explanation;
44      }
45      return SAT;
46  }
```

Figure 1: Theory Solver Algorithm

the edges of G_0 are zero, the difference between the values of two variables in the same SCC of G_0 is the same in all solutions to the constraints. In fact, each cycle in G_0 is of length 0 [17]; hence, if x and y are on one cycle of G_0 and the distance from x to y along the cycle is k, then the distance from y to x is $-k$. It follows that every solution to I must satisfy $y - x \leq k$ and $x - y \leq -k$, that is, $y - x = k$. In other words, a maximal SCC of G such that its vertex set induces also a maximal SCC of G_0 has only one family of solutions. (See Sect. 3.)

Each disequality constraint $d \in D$ is checked for feasibility against each SCC of G_0. If the two variables x, y in $x - y \neq n$ (where n may be 0) are in the same SCC of G_0 and $\delta(x) - \delta(y) = n$, then the set $I \cup Q \cup D$ is infeasible. The violated disequality d, together with the cycle that contains x and y and an appropriate subset of Q constitutes the explanation of infeasibility. If the two variables x and y in d are in the same SCC and $\delta(x) - \delta(y) \neq n$, then d is dropped from the set. Disequalities connecting variables in different SCCs of G_0 are simply passed on to the next phase of

```
47  GenerateBoundsForEachVariableInScc (scc′) {
48      x = FixValueOfOneVertexInScc (scc′);
49      U = ComputeUpperBoundForEachVariableInScc (scc′,x);
50      L = ComputeLowerBoundForEachVariableInScc (scc′,x);
51      return (L, U);
52  }

53  ComputeUpperBoundForEachVariableInScc (scc′,x) {
54      return BellmanFordAlgorithmWithFixedVertex (scc′,x);
55  }

56  ComputeLowerBoundForEachVariableInScc (scc′,x) {
57      rev = ReverseDirectionOfEdgesInScc (scc′);
58      return BellmanFordAlgorithmWithFixedVertex (rev,x);
59  }

60  CheckFeasibilityOfBoundsWithClique (SCC′, D, L, U) {
61      V = GatherVariablesInDisequalityConstraints(D);
62      Γ = GatherVariablesWithSameBounds (D, L, U);
63      ρ = GetBoundForGatheredVariables (Γ);
64      D′ = CollectRelevantDisequalityConstrints (D,Γ);
65      Γ′ = RemoveIrrelevantVariableByCheckingDegree (Γ, D′);
66      if (n(Γ′) ≤ ρ and n(V) = n(Γ)) return PROB_SAT;
67      else if (n(Γ′) ≤ ρ and n(V) ≠ n(Γ)) return UNDECIDED;
68      if (n(D′) < ((n(ρ) · (n(ρ) + 1))/2 and n(V) = n(Γ)) return PROB_SAT;
69      else if (n(D′) < ((n(ρ) · (n(ρ) + 1))/2 and n(V) ≠ n(Γ)) return UNDECIDED;
70      C = GenerateMaxClique (Γ′, D′);
71      V′ = GetVariablesInMaxClique (C);
72      if (n(V′) < ρ and n(V) = n(Γ)) return PROB_SAT;
73      else if (n(V′) < ρ and n(V) ≠ n(Γ)) return UNDECIDED;
74      else return GenerateExplanationFromMaxClique (SCC′,C);
75  }

76  SmallDomainEncodingForConstraintsInScc (scc′, D) {
77      CNF = InitializeCNF ();
78      CNF = CNF∪ EncodingForBoundsOfEachVariableInScc (scc′);
79      CNF = CNF∪ EncodingForInequalityConstraintsInScc (scc′);
80      CNF = CNF∪ EncodingForDisequalityConstraints (D);
81      return CNF;
82  }
```

Figure 2: Theory Solver Algorithm (continued)

the procedure. If no infeasibility is detected with G_0, a final feasibility check is performed by the small domain encoding method discussed in Sect. 3. For each SCC of G, Theorem 1 is used to compute bounds for each variable as follows.

To compute the upper bound for each variable, a variable in the SCC is chosen arbitrarily as source. (Variable x in Theorem 1.) The distance from it is computed for each variable in the SCC by the Bellman-Ford algorithm. The lower bound for a variable is computed as its distance from the same source variable used to compute the upper bound after reversing the edges in the SCC. (Note that one cannot replace the distances computed by these invocations of the shortest path algorithm with those computed on G_a.)

Some inequalities and disequalities may be automatically satisfied for all values of the variables in their ranges. For instance, if $0 \leq x \leq 1$ and $2 \leq y \leq 3$, then $x \neq y$ and $y - x \leq 4$ are both satisfied. These constraints are therefore ignored in the successive steps, which consist of a quick check based on finding a clique of the disequality graph, possibly followed by propositional encoding and satisfiability check.

The quick check is based on two observations: The first is that if all variables in the SCC have the same range, then the disequalities define a graph whose chromatic number must not exceed the size of the range for the constraints to be satisfiable. (The chromatic number is the least number of colors needed to assign different colors to adjacent vertices in the graph.) The second observation is that the chromatic number of a graph is bounded from below by the size of a clique of the graph and from above by the number of vertices. The process is described in lines 60–75 of Fig. 2. We identify sets of variables that have the same bounds and we check whether there are enough disequalities for the variables in one such set to cause inconsistency. Specifically, suppose a set $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ of variables is found such that all variables in Γ have the same bounds y_l and y_u. Let $\rho = y_u - y_l + 1$ be the range of each variable in Γ. If $p < \rho$ disequalities cannot cause inconsistency of this set of variables. If, on the other hand, the number of variables exceeds their common range, we check whether the disequalities form a clique of size greater than ρ. We first eliminate from Γ all variables that appear in fewer than ρ disequalities of the form $\gamma_i \neq \gamma_j$

($\gamma_i, \gamma_j \in \Gamma$). If Γ is not empty after this process, we greedily grow a clique, adding every time the variable appearing the largest number of disequalities among the surviving members of Γ. This greedy algorithm does not always find the largest clique, but is fast and works well in practice. The check results in one of three results: A suitable clique has been found and inconsistency is declared; a large enough clique was not found because of the heuristic nature of the algorithm; a large enough clique is known not to exist. In the first case, an explanation of inconsistency is derived from the disequalities forming the clique and the inequalities responsible for the bounds. In the last two cases, the result is inconclusive, because the chromatic number of a graph can be arbitrarily larger than the size of even the largest cliques. However, if a large enough clique does not exist in the graph, and the assignment is partial, we avoid a full check for inconsistency, which is rather expensive and likely to fail. (If the assignments to the boolean variables are complete, on the other hand, the consistency check must be performed for the whole decision procedure to be sound.)

In the final step of the theory solver, the constraints and the bounds are converted to a set of clauses whose satisfiability is established by calling a propositional SAT solver.[3] If the clauses are satisfiable, an assignment for the integer variables is extracted from the solution. Otherwise, an explanation for the unsatisfiability is derived as follows from the proof of unsatisfiability returned by the SAT solver, which consists of a subset of the clauses that are found to be unsatisfiable. (The *unsatisfiable core*.)

Every propositional clause is derived from some arithmetic constraint. If a clause appears in the unsatisfiable core, the parent constraint is included in the explanation. The bound constraints on the integer variables also contribute to unsatisfiability. They are accounted for by including all constraints that form the two shortest path spanning trees found during the computation of the bounds.

5. Related Work

Propositional abstraction as an approach to satisfiability modulo theories was proposed in [2]. Notable solvers based on that principle are MathSAT [4, 3], ICS [8], Verifun [10], BarcelogicTools [12, 21], SLICE [26], and SATORI [13]. ASAP [16] takes a dual approach, in which satisfiability of the propositional abstraction guarantees satisfiability of the original quantifier-free Presburger formula, while UCLID [18] is an eager solver. Our propositional enumeration engine is the one of [14, 15].

Finite instantiations for equality logic are studied in [22] and extended to difference logic in [25]; this last work has several points of contact with ours, but also important differences. The approach of [25] is eager, and the ranges are computed once and for all before invoking the propositional SAT solver. In contrast, we advocate a lazy approach and a computation of the ranges that takes place in the theory solver. Because of that, we may compute ranges more than once, but the size of the range for each variable in our algorithm is bounded by the sum of the slacks in the SCC, which is much smaller than $n + maxC$, where $maxC$ is the sum of absolute constants in the formula. In practice, ranges are much smaller in our algorithm. Moreover, we compute ranges by simply finding shortest paths in the constraint graph. The algorithm of [25], on the other hand, enumerates paths in the constraint graph and is exponential in the worst case.

Recent work by Ganai *et al.* [11] presents a polynomial algorithm for the computation of ranges, which improves over the one of [25], but shares the basic approach: ranges are allocated initially, so as to be adequate for every formula built from the given set of difference constraints. Disequalities are converted to disjunctions of inequalities, instead of being retained as such in the formulation of the problem. The theory consistency problem is never converted to propositional satisfiability. Instead, range propagation allows the solver to refine the initial ranges.

MathSAT introduced the notion of layered, incremental theory solver, and that of delayed theory combination; DPLL(t) the idea of exhaustive theory propagation, both of which are included in our implementation. The importance of considering zero-slack SCCs was first pointed out in [17], which deals with RDL. Finally, [26] discusses an efficient way to implement a recursive, backtrackable Bellman-Ford algorithm.

6. Experimental Results

We have implemented the algorithm presented in Sect. 4 in Sateen, a theorem prover for quantifier-free first-order logic that combines the propositional reasoning engine of [14, 15] with theory-specific procedures. A first set of experiments were done with the full set of QF_IDL (Quantifier free integer difference logic) benchmarks from SMT-COMP (Satisfiability Modulo Theories Competition [24]). The experiments were performed on a 1.7 GHz Pentium 4 with 2 GB of RAM running Linux. Time out was set at 3600 seconds. Sateen was compared with BarcelogicTools [9], Yices-0.1.1 [27] and MathSAT 3.3.1 [19]. The compared solvers are the ones that were submitted to SMT-COMP in 2005.

Figures 3–5 show scatterplots comparing BarcelogicTools, Yices and MathSAT to Sateen. Points below the diagonal represent wins for Sateen. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^{\eta}$, where κ and η are obtained by least-square fitting. Figure 3 shows that Sateen is comparable to BarcelogicTools. In Figures 4 and 5, Sateen shows better results compared to Yices and MathSAT, especially on hard problems. The SMT-COMP benchmark formulae are such that usually the sets of constraints passed to the theory solver either contain few disequality constraints, or are such that the disequality constraints are dealt with by the zero-slack SCC algorithm. The main purpose of these experiments is therefore not to show the effectiveness of the newly proposed algorithm for finite instantiations, but to establish that Sateen is, overall, a competent solver for IDL, comparable to some of the best tools in the field.

To assess the effectiveness of the finite instantiation approach, we have generated two benchmark suites where disequality constraints play a significant role: the Queens Suite and the Job Shop Scheduling Suite. The Queens Suite contains n-Queens problem and n-Super-Queens problem. The n-Queens problem consists of placing n queens on a $n \times n$ board so that they do not attack each other. In the n-Super-Queens problem, each queen's placement is more restricted by allowing it also the knight's moves. The Job Shop Scheduling problem checks the feasibility of processing a number of jobs, each consisting of several tasks, on a given set of machines in a given amount of time. These two sets of benchmarks have disequality constraints that cause pigeonholing problems. In the experiment on these benchmarks, the timeout was set to 1000 seconds.

Figures 6–8 shows that Sateen is often orders of magnitude faster than the other solvers on these problems. The symbol $\times$ represents the experiment on the Queens benchmark, and the symbol $+$ represents the experiment on the Job Shop Scheduling benchmark. We also provide the comparison between Sateen with our proposed algorithm and a version of Sateen that splits disequalities. Figure 9 shows that the finite instantiation algorithm works significantly bet-

[3] Our current encoding of the ranges is rather unsophisticated. We are implementing a heuristic approach to minimizing the total number of encoding bits required.

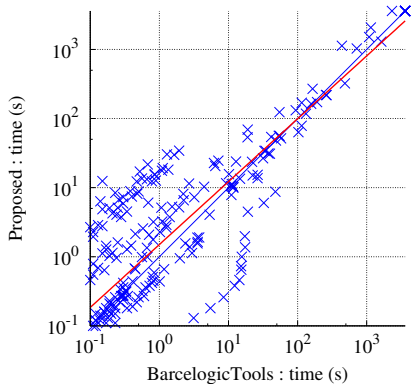


Figure 3: BARCELOGICTOOLS vs. Sateen on QF_IDL

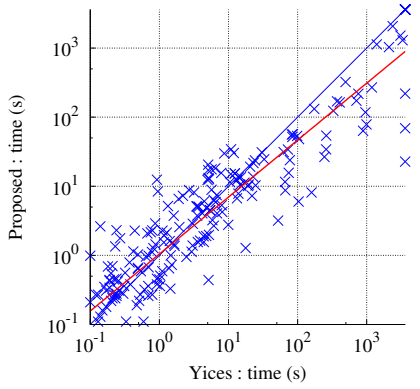


Figure 4: YICES vs. Sateen on QF_IDL

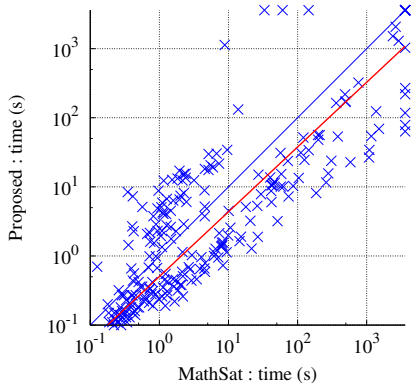


Figure 5: MATHSAT vs. Sateen on QF_IDL

ter than the splitting method. The clique detection algorithm is particularly helpful in Job Shop Scheduling problems.

7. Conclusions

We have presented an approach to solving integer difference logic that is particularly effective when the constraints to be solved are rich in disequalities. By restricting consideration to a small sufficient set of solutions, we are able to compute bounds for the integer variables occurring in the constraints. Experiments indicate that this approach is more effective than splitting disequalities into the disjunction of inequalities. Further improvements in efficiency are

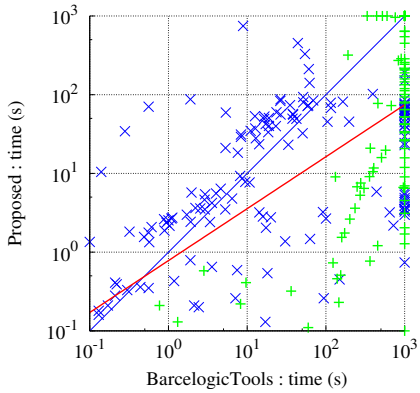


Figure 6: BARCELOGICTOOLS vs. Sateen on Job Shop Scheduling and Queen Suites

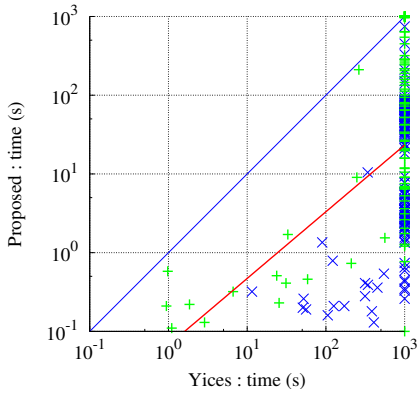


Figure 7: YICES vs. Sateen on Job Shop Scheduling and Queen Suites

expected from a more sophisticated encoding scheme for the finite instances that we are currently developing.

Acknowledgment. The authors thank Alberto Oliveras, Alessandro Cimatti, and Leonardo deMoura for their help with BarcelogicTools, MathSAT, and Yices.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, Snowbird, UT, June 2001.

[2] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 236–249. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[3] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Seventeenth Conference on Computer Aided Verification (CAV'05)*, pages 335–349. Springer-Verlag, Berlin, July 2005. LNCS 3576.

[4] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *International Conference on Tools and Algorithms*

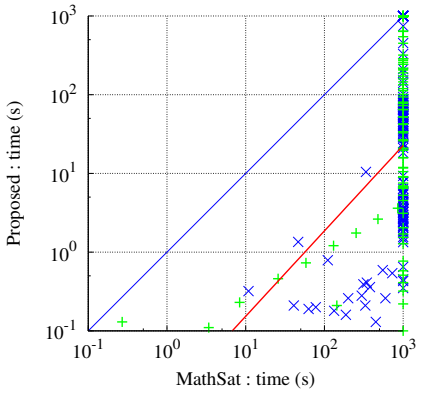


Figure 8: MATHSAT vs. Sateen on Job Shop Scheduling and Queen Suites

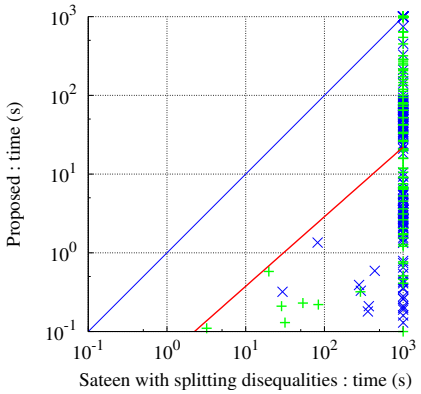


Figure 9: Sateen with splitting disequalities vs. Sateen on Job Shop Scheduling and Queen Suites

for Construction and Analysis of Systems (TACAS'05), pages 317–333, Edinburgh, UK, Apr. 2005. LNCS 3440.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *An Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[8] L. de Moura and H. Reuß. Lemmas on demand for satisfiability solvers. In *Fifth International Symposium on the Theory and Application of Satisfiability Testing (SAT'02)*, Cincinnati, OH, May 2002.

[9] Url: http://www.lsi.upc.edu/ oliveras/bclt-main.html.

[10] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In W. A. Hunt, Jr. and F. Somenzi, editors, *Fifteenth Conference on Computer Aided Verification (CAV'03)*, pages 355–367. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[11] M. K. Ganay, M. Talupur, and A. Gupta. SDSAT: Tight integration of small domain encoding and lazy abstraction in a separation logic solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06)*, pages 135–150, Vienna, Austria, Mar. 2006. LNCS 3920.

[12] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 175–188. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[13] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. SATORI – a fast sequential SAT engine for circuits. In *Proceedings of the International Conference on Computer-Aided Design*, pages 320–325, San Jose, CA, Nov. 2003.

[14] H. Jin, H. Han, and F. Somenzi. Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05)*, pages 287–300, Apr. 2005. LNCS 3440.

[15] H. Jin and F. Somenzi. Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In *Proceedings of the Design Automation Conference*, pages 750–753, Anaheim, CA, June 2005.

[16] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 308–320. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[17] S. Lahiri and M. Musuvathi. An efficient Nelson-Oppen decision procedure for difference constraints over rationals. In *Third International Workshop on Pragmatical Aspects of Decision Procedures in Automated Reasoning (PDPAR'05)*, pages 2–9, Edinburgh, UK, July 2005. To appear in ENTCS.

[18] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 475–478. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[19] Url: http://mathsat.itc.it.

[20] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[21] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Seventeenth Conference on Computer Aided Verification (CAV'05)*, pages 321–334. Springer-Verlag, Berlin, July 2005. LNCS 3576.

[22] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Journal of Information and Computation*, 178(1):279–293, Oct. 2002.

[23] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

[24] Url: http://www.csl.sri.com/users/demoura/smt-comp/.

[25] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 148–161. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[26] C. Wang, F. Ivancic, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR'2005)*, Montego Bay, Jamaica, Dec. 2005.

[27] Url: http://fm.csl.sri.com/yices.

Tracking MUSes and Strict Inconsistent Covers

Éric Grégoire Bertrand Mazure Cédric Piette
CRIL-CNRS & IRCICA
Université d'Artois
rue Jean Souvraz SP18
F-62307 Lens Cedex France
{gregoire,mazure,piette}@cril.univ-artois.fr

Abstract

In this paper, a new heuristic-based approach is introduced to extract minimally unsatisfiable subformulas (in short, MUSes) of SAT instances. It is shown that it often outperforms current competing methods. Then, the focus is on inconsistent covers, which represent sets of MUSes that cover enough independent sources of infeasibility for the instance to regain satisfiability if they were repaired. As the number of MUSes can be exponential with respect to the size of the instance, it is shown that such a concept is often a viable trade-off since it does not require us to compute all MUSes but provides us with enough mutually independent infeasibility causes that need to be addressed in order to restore satisfiability.

1. Introduction

In this paper, the focus is on computational approaches to detect minimal unsatisfiable subformulas (MUSes) of unsatisfiable SAT instances. Detecting MUSes can prove valuable in many applications. For example, when we check the consistency of knowledge-bases, we prefer knowing which clauses are contradicting one another rather than only knowing that the whole base is inconsistent. MUSes provide such an information, as they represent the smallest explanations (in terms of the number of involved clauses) for unsatisfiability.

Unfortunately, computing MUSes exhibits a high worst-case complexity. Indeed, checking whether a set of clauses is a MUS is DP-complete [21], and checking whether a formula belongs to the set of MUSes of an unsatisfiable instance or not, is in Σ_2^p [9]. Moreover, the number of MUSes can be exponential in the size of the instance. Indeed, the number of MUSes of an n-clauses instance is $C_n^{n/2}$ in the worst case. However, let us stress that the number of MUSes remains often tractable in real-life situations. For example, in model-based diagnosis [13], based on experimental studies, it is often assumed that single faults occur, which is often translated by a limited number of MUSes.

Recently, several approaches have been proposed to approximate or compute MUSes. Unfortunately, they concern specific classes of clauses or they remain tractable for small instances, only. Among them, let us mention Bruni's work [5], who has shown how a MUS can be extracted in polynomial time through linear programming techniques for clauses exhibiting a so-called integral point property. However, only restrictive classes of clauses obey such a property (mainly Horn, renameable Horn, extended Horn, balanced and matched ones). Other studies about the complexity and algorithmic properties of extracting MUSes for specific classes of clauses can be found in [6, 7] and [10]. In [4], Bruni has also proposed an approach that approximates MUSes by means of an adaptative search guided by clauses hardness. Zhang and Malik have described in [23] a way to extract MUSes by learning nogoods involved in the derivation of the empty clause by resolution. In [17], Lynce and Marques-Silva have proposed a complete and exhaustive technique to extract smallest MUSes. Oh and her co-authors have presented in [20] a Davis, Putnam, Logemann and Loveland DPLL-oriented approach that is based on a marked clauses concept to allow one to approximate MUSes. Liffiton and Sakallah have shown how MUSes can be computed through the dual concept of maximally satisfiable subsets [16].

In this paper, a new heuristic-based approach to approximate and compute MUSes is introduced. It is based on a concept of *critical clauses w.r.t. an inter-*

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

pretation that allows us to refine the approach by [19] to locate approximations of MUSes. Although it exploits a heuristic information, let us stress that the approach is complete in the sense that it always delivers a MUS for any unsatisfiable instance. Then, a concept of *inconsistent cover* is introduced. Inconsistent covers represent sets of MUSes that cover enough independent sources of infeasibility that would allow the instance to regain satisfiability if they were fixed. As the number of MUSes can be exponential with respect to the size of the instance, such a concept can be a viable trade-off since it does not require us to compute all MUSes but provides us with enough infeasibility causes that would allow the instance to become satisfiable if they were all repaired.

The paper is organized as follows. In the next section, the concepts of MUS and inconsistent cover are presented formally. In section 3, the crucial notion of critical clause w.r.t. an interpretation is introduced and analyzed. In section 4, the new approach to approximate or compute one MUS is presented. Extensive experimental results are given in section 5. Before we conclude, section 6 shows how the approach can be extended to compute strict inconsistent covers.

2. MUSes and Inconsistent Covers

Let $\mathcal{L}$ be a standard Boolean logical language built on a finite set of Boolean variables, denoted a, b, etc. Formulas will be denoted using upper-case letters such as C. Sets of formulas will be represented using Greek letters like Γ or Σ. An interpretation is a truth assignment function that assigns values from $\{true, false\}$ to every Boolean variable.

A formula is consistent or satisfiable when there is at least one interpretation that satisfies it, i.e. that makes it become *true*. An interpretation will be denoted by upper-case letters like I and will be represented by the set of literals that it satisfies. Actually, any formula in $\mathcal{L}$ can be represented (while preserving satisfiability) using a set (interpreted as a conjunction) of clauses, where a clause is a finite disjunction of literals and where a literal is Boolean variable that can be negated. SAT is the canonical NP-complete problem that consists in checking whether a set of Boolean clauses is satisfiable or not, i.e. whether there exists an interpretation that satisfies all clauses in the set or not. Let us also recall the SAT-related optimization problem, namely max-SAT.

Definition 1 *Given a SAT instance Γ, max-SAT consists in finding the maximum number of clauses of Σ that can be satisfied under a same interpretation.*

When a SAT instance is unsatisfiable, it exhibits at least one minimally unsatisfiable subformula, in short one *MUS*.

Definition 2 *A MUS Γ of a SAT instance Σ is a set of clauses s.t.*

1. $\Gamma \subseteq \Sigma$
2. Γ *is unsatisfiable*
3. *Every proper subset of* Γ *is satisfiable*

In the following, another crucial concept is the notion of *(strict) inconsistent cover*.

Definition 3 *Two sets of clauses are independent if and only if their intersection is empty. One (strict) inconsistent cover IC of an unsatisfiable SAT instance Σ is a set-theoretic union of (independent) MUSes of Σ s.t. $\Sigma \backslash IC$ is satisfiable.*

It should be noted that a same unsatisfiable instance can exhibit several different strict inconsistent covers. For example, let $\Sigma = \{\neg a, \neg b, a \vee b, c, \neg d \vee b, \neg c \vee a, d\}$. Σ contains 3 MUSes: $MUS_1 = \{a \vee b, \neg a, \neg b\}$, $MUS_2 = \{d, \neg b, \neg d \vee b\}$ and $MUS_3 = \{c, \neg c \vee a, \neg a\}$. Σ contains 2 strict inconsistent covers, namely $IC_1 = MUS_1$ and $IC_2 = MUS_2 \cup MUS_3$.

Although a strict inconsistent cover does not provide us with the set of all MUSes that may be present in a formula, it gives us a series of minimal explanations for infeasibility that are sufficient to explain and potentially repair enough sources of unsatisfiability in order for the whole formula to regain satisfiability.

A straightforward result is that a strict inconsistent cover enables us to get a lower-bound of the number of unsatisfied clauses in a max-SAT solution of an unsatisfiable SAT instance.

Property 1 *Let Σ be an unsatisfiable SAT instance. Let IC be a strict inconsistent cover of Σ. Let $|IC|$ be the number of independent MUSes contained in IC. For any interpretation I of Σ, at least $|IC|$ clauses of Σ are falsified under I.*

3. A New Heuristic to Detect MUSes

In [19] it is shown how local search can be helpful for approximating MUSes. The basic idea is that clauses that are often falsified during a failed local search for satisfiability belong most probably to MUSes, when the instance is actually unsatisfiable. When the score of a clause is the number of times it has been falsified during

a failed local search (in short, failed LS), discriminating the clauses with a high score can deliver a good approximation of the set of MUSes. Such a heuristic has been studied in an extensive manner in [18] and [19]. It has also been extended in several ways to address decision and optimization problems that belong to higher levels of the polynomial hierarchy (see e.g. [11, 3, 12] and [2]).

In the following, we assume that the SAT instance is unsatisfiable. The above heuristic can require us to increment the score of clauses even when they do not actually belong to any MUS. Unless we solve the problem of finding MUSes itself, we can only rely on some heuristic indications about the extent to which a currently falsified clause could or could not belong to a MUS. In this respect, we claim that some relevant parts of the neighborhood of the current interpretation can be checked and provide more information about whether a currently falsified clause C should be counted or not. The idea is to take the structure of C into account and to increment the score of C only when it cannot be satisfied without conducting other clauses to be falsified in their turn. We shall see that this technique implements definitions that approximate a proposition that is intrinsic to clauses belonging to MUSes.

To illustrate this concept, let us use the following example. Let $\Delta = \{a \vee b \vee c, \neg a \vee b, \neg b \vee c, \neg c \vee a, \neg a \vee \neg b \vee \neg c\}$. Δ is unsatisfiable and is its own MUS. Let $I = \{a, b, c\}$ an interpretation. Under this interpretation, only the $\neg a \vee \neg b \vee \neg c$ clause is falsified. In the following, the *once-satisfied* clause concept will prove useful.

Definition 4 *A clause C is once-satisfied by an interpretation I if and only if exactly only one literal of C is satisfied under I.*

In the above example, the clauses $\neg a \vee b$, $\neg b \vee c$ and $\neg c \vee a$ are once-satisfied by $I = \{a, b, c\}$.

Definition 5 *A clause C falsified under an interpretation I is critical w.r.t. I if and only if the opposite of every literal of C belongs to a clause that is once-satisfied by I. These once-satisfied clauses that are not tautological ones are called linked to C.*

In the example, $\neg a \vee \neg b \vee \neg c$ is falsified under I and is critical w.r.t. I; its related linked clauses are the once-satisfied ones $\neg a \vee b$, $\neg b \vee c$ and $\neg c \vee a$.

The role of these definitions is easily understood thanks to the following property.

Property 2 *Let C be a critical clause w.r.t. an interpretation I, then any flip from I to I' such that C is satisfied under I' will conduct I' to falsify at least one clause that was satisfied under I.*

In order to discriminate clauses belonging to MUSes, the idea is to increment the scores of critical clauses during the search, together with their linked (satisfied) clauses, rather than increment the scores of all falsified clauses. Such a technique can be easily grafted to a LS algorithm and the updates can be easily computed. Actually, it implements definitions that approximate a property that is obeyed by clauses belonging to MUSes.

Property 3 *Let I be an interpretation giving an optimal result for max-SAT on an unsatisfiable instance Σ. Then, any falsified clause C w.r.t. I belongs to at least one MUS of Σ and is critical w.r.t. I. Moreover, at least one once-satisfied clause linked to C also belongs to a MUS of Σ.*

Our technique is thus an approximation one in the sense that clauses and their linked ones are considered during the whole search, and not w.r.t. interpretations that are solutions of max-SAT. Indeed, being a critical clause is neither a necessary nor a sufficient condition to belong to a MUS. As the following example illustrates, a critical clause w.r.t. an interpretation that is not an optimal one w.r.t. max-SAT for an unsatisfiable formula might not belong to a MUS. Let $\Delta = \{a \vee d, \neg a \vee \neg b, \neg d \vee e, f, \neg e \vee \neg f\}$. Clearly, Δ is satisfiable. $\neg e \vee \neg f$ is falsified under $I = \{a, b, d, e, f\}$ and is critical w.r.t. I. Moreover, a clause from a MUS that is falsified under a given interpretation I is not necessary critical w.r.t. I, as the following example shows. Let $\Delta = \{a \vee d, b, \neg a \vee \neg b, \neg d \vee e, f, \neg e \vee \neg f\}$. Clearly, Δ is a minimal unsatisfiable set of clauses. $\neg a \vee \neg b$ is falsified under $I = \{a, b, d, e, f\}$. However, it is not critical w.r.t. I. Fortunately, the following property ensures that all clauses from a MUS can be scored by the heuristic.

Property 4 *Let Γ be a MUS. For all clauses $C \in \Gamma$, there exists an interpretation I s.t. C is critical w.r.t. I.*

This property ensures that any clause that takes part to a MUS can be critical w.r.t. at least one interpretation. As such, this property does not guarantee that our scoring heuristic will allow us to exhibit all clauses belonging to MUSes. Indeed, it does not indicate that a LS run will necessary increment the score of all such clauses at least once since LS does not necessary visit all interpretations. However, the following property and its corollary provide us with a good indication that LS will probably visit interpretations

where clauses belonging to MUSes are critical. Indeed, it is well-known that LS is in general attracted by local minima. Property 5 ensures that all falsified clauses are critical in local or global minima.

Definition 6 *A local minimum is an interpretation s.t. no flip can increase the number satisfied clauses. A global minimum (or max-SAT solution) is an interpretation delivering the maximal number of satisfied clauses.*

Property 5 *In (local or global) minima, all falsified clauses are critical.*

A corollary ensures that at least one clause per MUS is critical in such minima.

Corollary 1 *In (local or global) minima, at least one clause per MUS is critical.*

4. Approximating and Computing one MUS

In the following, it is shown that a meta-heuristic based on scoring critical clauses is viable in order to approximate or compute MUSes. Actually, due to implementation efficiency constraints, we update the scores of critical clauses only. Updating the scores of their linked clauses does not lead to dramatic performance improvements, at least w.r.t. our selected LS algorithm and tested benchmarks.

The main idea is as follows. Let Σ be an unsatisfiable SAT instance. While local search fails to find a model of Σ, we remove clauses of Σ with the lowest scores. We record the obtained sub-formulas on a stack. Next, the unsatisfiability of the last subformula where LS fails to find a model is checked. If this subformula is unsatisfiable, then it is an approximation of a MUS of Σ. Otherwise, this unsatisfiability test is repeated on the lastly recorded supersets of clauses, until one of them is proved unsatisfiable. This algorithm, called `AOMUS` (Approximate One MUS), is described in Algorithm 1.

Then an exact MUS can be obtained by a step-by-step minimization of the upper-approximation until the remaining clauses are proven to form a MUS (see [14] for an alternative method). This process is called `fine-tune` (see Algorithm 2). The order of tested clauses can be guided by the score of each clause.

Let us stress that this algorithm is complete in the sense that it always delivers one MUS for any unsatisfiable instance. The combination of `AOMUS` algorithm and `fine-tune` procedure is called `OMUS` (find One MUS) and is described in Algorithm 3.

Algorithm 1: `AOMUS` algorithm

```
Input: an unsatisfiable SAT formula Σ
Output: an Approximation of One MUS of Σ
1  begin
2      stack ⟵ ∅ ;
3      while (LS + Score(Σ) fails to find a model) do
4          push(Σ) ;
5          Σ ⟵ Σ \ LowestScore(Σ) ;
6      repeat
7          Σ ⟵ pop() ;
8      until Σ is UNSAT ;
9      return Σ ;
10 end
```

Algorithm 2: `fine-tune` procedure

```
Input: an approximation of a MUS of Σ
Output: a MUS extracted from Σ
1  begin
2      foreach clause c ∈ Σ sorted w.r.t. their scores do
3          if (Σ \ {c} is unsatisfiable) then
4              Σ ⟵ Σ \ {c} ;
5      return Σ ;
6  end
```

Its efficiency directly depends on the quality of the upper-approximation. In the next section, experimental results show that the approximation delivered by `AOMUS` is often of a good quality, because a very small set of clauses is removed by the `fine-tune` step and in consequence a very small number of unsatisfiability tests are performed (when a clause belongs to the MUS, the test amounts to a consistency check).

Actually, we refined this basic procedure in the following manner. Assume that the current computed subformula is actually unsatisfiable; whenever a unique clause remains falsified during the LS run, we are sure that this clause belongs to all MUSes of this current subformula.

We mark these clauses as *protected* and they cannot be removed from Σ thereafter. Moreover, this information is kept all along the process because it can prove very useful during the minimization procedure. After the approximation is computed, the idea is to remove each clause to verify whether it participates to the cause of unsatisfiability of the formula or not. However, protected clauses do not need to be tested, because we know that removing one of them restores satisfiability. Furthermore, when the remaining falsified clauses contain protected clauses only, they form one exact MUS. In this case, the fine-tune step can be omitted since an exact MUS has been already extracted

Algorithm 3: OMUS algorithm

Input: an unsatisfiable SAT formula Σ
Output: One MUS of Σ

```
1 begin
2 |  Σ ⟵ AOMUS(Σ) ;
3 |  Σ ⟵ fine-tune(Σ) ;
4 |  return Σ ;
5 end
```

from the formula.

It appears that this refinement proves useful for many instances, and allows dramatic efficiency gains for both AOMUS and OMUS algorithms.

The parameters that were selected for these methods are as follows. As a case study, Wsat [15] with the *Rnovelty+* option was chosen as the LS procedure. The following parameters were fine-tuned based on extensive tests on various benchmarks. After each flip of the LS, the score of critical clauses is increased by the number of their linked clauses. This technique allows us to take the length of critical clauses into account, since the number of linked clauses depends on the length of the critical clause in terms of the number of involved literals. Now, clauses whose score is lower than $(\textit{min-score}+\frac{\#Flips}{\#Clauses})$ are dropped, where *min-score* is the lowest score for a clause of Σ; $\#Flips$ and $\#Clauses$ are the number of performed flips and the number of clauses in Σ, respectively. This procedure was tested extensively on various UNSAT instances from several difficult benchmarks from DIMACS [8] and from the annual SAT competitions [22], and compared with other published approaches to compute MUSes, as described in the next section.

5. Experimental Results

All experiments have been conducted on Pentium IV, 3Ghz under linux Fedora Core 4. As our results show, this approximation delivers an exact result most of the time. Moreover, the fine-tune procedure ensures that a MUS is actually obtained. As most current approaches do not guarantee that the delivered unsatisfiable sets of clauses are actually MUSes, we provide both the results of applying AOMUS and OMUS. However, on many instances, approximations delivered thanks to AOMUS appeared to be actual MUSes. Moreover, very often, the last subformula where LS fails to find a model is in fact unsatisfiable. Thus, in practice, the last loop of the AOMUS algorithm reduces to a unique inconsistency test, most of the time.

We compared our approach with an adaptation of AOMUS where *Score* is the basic heuristic of [19], which simply counts the number of times a clause is falsified. We also compared our approach with zCore, the core extractor of zChaff [23]. zChaff is currently one of the most efficient SAT solvers. We also ran Lynce and Marques-Silva's procedure [17], and took Bruni's [4] experimental results into account. For Bruni's technique, we only mention the experimental results obtained by the author, since this system is not available. Although a comparison with Bruni's technique is thus difficult to achieve from an experimental side, it appears that Bruni's technique has been experimented on small instances, only. zCore proved competitive for single-MUS instances but failed to deliver good results when several MUSes are present. Indeed, zCore does not concentrate on finding one MUS, but on finding proofs of unsatisfiability. Not surprisingly, our approach proved more efficient than the similar one where *Score* is based on [19] heuristic. Most often, it proved to be more competitive than all the other considered techniques when very large and difficult multi-MUSes instances were considered. Noticeably, it was also the only technique to perform in a competitive way on all benchmarks. Let us stress that the Lynce-Silva's procedure computes the smallest MUS, that zCore delivers an approximation of a MUS, whereas our OMUS and AOMUS procedures deliver one exact and one approximate MUS, respectively. Moreover, it should be emphasized that MUSes that are discovered by the various approaches are not necessary the same ones.

In Table 1, some typical experimental results are given. Except for Bruni's results which are just size results that we have extracted from [4], we provide both the experimental size of the discovered smallest unsatisfiable subsets, together with the CPU time in seconds to get them. Time-out indicates that no result has been obtained within 1 hour CPU time. For example, for the homer14 instance, AOMUS delivered an approximate MUS made of 561 clauses within 28.03 s. Actually, this was an exact MUS, as it was found by OMUS in 30.64 s. Note that an AOMUS version based on [19] delivered the same result in 347.19 s. zCore delivered an approximate MUS made of 1065 clauses within 714 s. Actually, this approximate MUS was a superset of the MUS discovered by both AOMUS and OMUS. Also, it can be seen e.g. on the fpga benchmarks that AOMUS (i.e. our approach without the fine-tune procedure) delivered smaller unsatisfiable subsets than any other considered method, most often. Let us also emphasize that even on small instances like the aim ones, OMUS proved very competitive, as well.

Table 1. Experimental results: Approximate One MUS (AOMUS) and find One MUS (OMUS)

Instance	#var	#cla	Lynce&Silva [17]		Bruni [5]	zCore [23]		Scoring like [19]		AOMUS		OMUS	
			#cla	Time	#cla	#cla	Time	#cla	Time	#cla	Time	#cla	Time
fpga10_11	220	1122	Time out		-	561	28.51	561	18.26	561	13.06	561	13.75
fpga10_12	240	1344	Time out		-	672	71.27	561	30.11	561	16.9	561	17.03
fpga10_13	260	1586	Time out		-	793	166.99	561	51.67	561	25.95	561	31.89
fpga10_15	300	2130	Time out		-	1065	570.3	561	128.05	561	44.18	561	68.17
fpga11_12	264	1476	Time out		-	738	112.53	738	66.8	738	65.49	738	66.3
fpga11_13	286	1742	Time out		-	871	504.97	738	180.66	738	56.71	738	84.74
fpga11_14	308	2030	Time out		-	1015	1565.6	738	415.32	738	69.55	738	304.4
fpga11_15	330	2340	Time out		-	Time out		738	568.79	738	52.14	738	85.2
aim100-1_6-no-2	100	160	53	224	54	54	0.05	53	0.268	53	0.38	53	0.38
aim100-2_0-no-1	100	200	Time out		19	19	0.09	19	0.216	19	0.19	19	0.23
aim200-1_6-no-3	200	320	Time out		86	83	0.07	83	0.37	83	0.44	83	0.83
aim200-2_0-no-3	200	400	Time out		37	37	0.23	37	0.39	37	0.49	37	0.54
aim50-1_6-no-4	50	80	20	1.18	20	20	0.04	20	0.163	20	0.16	20	0.17
aim50-2_0-no-4	50	100	21	3.49	21	21	0.14	21	0.208	21	0.22	21	0.27
2bitadd_10	590	1422	Time out		-	815	343.48	1212	42.752	806	189.47	716	268.5
barrel2	50	159	Time out		-	77	0.04	100	0.35	77	0.36	77	0.44
jnh10	100	850	Time out		161	68	0.88	128	9.35	79	42.25	79	42.9
jnh20	100	850	Time out		120	102	0.23	104	21.68	87	48.93	87	75.76
jnh5	100	850	Time out		125	86	0.39	140	12.653	88	46.2	86	46.87
jnh8	100	850	Time out		91	90	0.22	162	28.964	69	90.53	67	99.07
homer06	180	830	Time out		-	415	15.96	415	10.97	415	9.04	415	9.3
homer07	198	1012	Time out		-	506	21.6	415	12.59	415	10.67	415	19.19
homer08	216	1212	Time out		-	606	44.46	554	23.43	415	19.79	415	24.65
homer09	270	1920	Time out		-	960	141.48	415	93.19	504	60.9	415	81.23
homer10	360	3460	Time out		-	940	624.11	1614	148.27	503	466.94	415	513.11
homer11	220	1122	Time out		-	561	23.44	561	41.68	561	15.6	561	16.32
homer12	240	1344	Time out		-	672	76.19	708	25.92	564	41.03	561	62.34
homer13	260	1586	Time out		-	793	152.13	579	67.38	561	76.66	561	78.51
homer14	300	2130	Time out		-	1065	714.03	561	347.19	561	28.03	561	30.64
homer15	400	3840	Time out		-	Time out		677	247.84	561	1048.28	561	1104.13
homer16	264	1476	Time out		-	738	115.49	738	78.44	738	61.31	738	62.91
homer17	286	1742	Time out		-	871	369.11	870	127.43	738	68.28	738	87.4

More extensive results can be downloaded from `http://www.cril.univ-artois.fr/~piette`

6. Extracting Strict Inconsistent Covers

These last years, several approaches have been proposed to compute *all* MUSes of a SAT instance ([1, 16]). Unfortunately, these computational approaches remain often intractable since, among other things, the number of MUSes in a formula can be exponential in the size of the formula. In this section, a novel method is introduced to compute *independent* MUSes, i.e. MUSes that do not share any clause. The idea motivating this approach is that independent MUSes express independent −uncorrelated− causes of infeasibility inside a same formula. This lead us to the concept of *strict inconsistent cover* of an unsatisfiable instance.

Although an inconsistent cover does not provide us with the set of all MUSes that may be present in a formula, it does however provide us with a series of minimal explanations for unsatisfiability that are sufficient to explain and potentially repair enough sources of infeasibility in order for the whole formula to regain satisfiability. Moreover, formulas can have MUSes that are very small with respect to the size of the formula; dropping or repairing such MUSes can sometimes be sufficient to regain satisfiability. Since strict inconsistent covers are composed of independent MUSes, one way to obtain a strict inconsistent cover is to compute a single MUS, then remove it from the formula, and repeat these operations until the remaining subformula becomes satisfiable. This method is described in the following ICMUS (find a strict Inconsistent Cover of MUSes) algorithm (see Algorithm 4).

Algorithm 4: ICMUS algorithm

Input: an unsatisfiable SAT formula Σ
Output: a strict Inconsistent Cover of Σ

```
1 begin
2 |  IC ⟵ ∅ ;
3 |  while (Σ is unsatisfiable) do
4 |  |  MUS ⟵ OMUS(Σ) ;
5 |  |  IC ⟵ IC ∪ MUS ;
6 |  |  Σ ⟵ Σ \ MUS ;
7 |  return IC ;
8 end
```

In Table 2, some typical experimental results are provided. Unsurprisingly, a lot of instances exhibit very small inconsistent covers in terms of the number of involved clauses. For example, let us consider ezfact16_2. This formula contains 1113 clauses, and

Table 2. Inconsistent covers for various classes of formulas

Instance	#var	#cla	Time	**#MUSes** in the IC	(#var,#cla) for each MUS
dp02u01	213	376	1.19	**1**	(47,51)
dp03u02	478	1007	362	**1**	(327,760)
23.cnf	198	474	2.68	**1**	(165,221)
42.cnf	378	904	9	**1**	(315,421)
fpga10_11_uns_rcr	220	1122	56	**2**	(110,561) (110,561)
fpga11_12_uns_rcr	264	1476	128	**2**	(132,738) (132,738)
ca002	26	70	0.61	**1**	(20,39)
ca004	60	168	1.11	**1**	(49,108)
ca008	130	370	5.26	**1**	(110,255)
term1_gr_rcs_w3	606	2518	6180	**11**	(12,22) (21,33) (30,58) (12,22) (12,22) (12,22) (12,22) (12,22) (12,22) (24,39) (21,33)
C220_FV_RZ_14	1728	4508	28	**1**	(10,14)
C220_FV_RZ_13	1728	4508	46	**1**	(9,13)
C170_FR_SZ_96	1659	4955	18	**1**	(81,233)
C208_FA_SZ_121	1608	5278	21	**1**	(18,32)
C168_FW_UT_851	1909	7491	83	**1**	(7,9)
C202_FW_UT_2814	2038	11352	304	**1**	(15,18)
jnh208	100	800	14	**1**	(76,119)
jnh302	100	900	63	**2**	(27,28) (98,208)
jnh310	100	900	184	**2**	(12,13) (90,188)
ezfact16_1	193	1113	203	**1**	(37,54)
ezfact16_2	193	1113	104	**1**	(32,41)
ezfact16_3	193	1113	207	**1**	(63,128)
3col40_5_3	80	346	4.64	**1**	(64,136)
fphp-012-010	120	1212	57	**1**	(120,670)

possesses one MUS that is composed of 41 clauses. This MUS is in fact a strict inconsistent cover because its removal restores the formula satisfiability. The industrial-related formula `term1_gr_rcs_w3` exhibits a strict inconsistent cover made of 11 small MUSes. Thanks to this result, we can deduce that all interpretations falsify at least 11 clauses. Indeed, as shown in Property 1 a strict inconsistent cover enables us to get an lower-bound of unsatisfied clauses in a max-SAT solution of an unsatisfiable instance. Indeed, the extracted MUSes do not share constraints and we know that all interpretations falsify at least one clause per MUS. Let us also note that inconsistent covers on `fpga` formulas suggest the presence of a form of symmetry in this type of instances.

7. Conclusion

In this paper, we have introduced a concept of clauses that are critical with respect to a given interpretation during a local search run. The main properties of this concept have been analyzed formally. We have shown that it plays a crucial role in a new heuristic-based approach to approximate or compute MUSes. Accordingly, our experimental results show the very good performance of this new approach which tracks MUSes according to the trace of a failed local search run for consistency checking. We have also introduced a strict inconsistent cover concept. This concept allows us to avoid computing all MUSes when the goal is to explain enough causes of unsatisfiability that would allow the instance to regain satisfiability if they were all repaired. Accordingly, our heuristic-based approach has been extended to address the search for inconsistent covers. Once again, the approach proves efficient on many difficult benchmarks. In the future, we plan to explore how the critical clause concept could be refined in order to further improve the performance of our algorithms.

Acknowledgement

This study has been supported by the EC under a FEDER grant and by the *Région Nord/Pas-de-Calais*. The authors thank the reviewers for their valuable comments.

References

[1] J. Bailey and P. J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *PADL*, pages 174–186, 2005.

[2] F. Boussemart, F. Hémery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150, Valencia (Spain), 2004.

[3] L. Brisoux, É. Grégoire, and L. Saïs. Checking depth-limited consistency and inconsistency in knowledge-based systems. *International Journal of Intelligent Systems*, 16(3):333–360, 2001.

[4] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Applied Mathematics*, 130(2):85–100, 2003.

[5] R. Bruni. On exact selection of minimally unsatisfiable subformulae. *Annals of Mathematics and Artificial Intelligence*, 43(1):35–50, 2005.

[6] H. Büning. On subclasses of minimal unsatisfiable formulas. *Discrete Applied Mathematics*, 107(1–3):83–98, 2000.

[7] G. Davydov, I. Davydova, and H. Büning. An efficient algorithm for the minimal unsatisfiability problem for a subclass of cnf. *Annals of Mathematics and Artificial Intelligence*, 23(3–4):229–245, 1998.

[8] DIMACS. Benchmarks on SAT. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/.

[9] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates and counterfactual. *Artificial Intelligence*, 57:227–270, 1992.

[10] H. Fleischner, O. Kullman, and S. Szeider. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, 289(1):503–516, 2002.

[11] É. Grégoire and D. Ansart. Overcoming the christmas tree syndrome. *International Journal on Artificial Intelligence Tools (IJAIT)*, 9(2):97–111, 2000.

[12] É. Grégoire, B. Mazure, and L. Saïs. Using failed local search for SAT as an oracle for tackling harder A.I. problems more efficiently. In *International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA'02)*, pages 51–60, LNCS 2443, Springer, Varna (Bulgaria), 2002.

[13] C. L. Hamscher W. and de Kleer J., editors. *Readings in Model-Based Diagnosis*. Morgan Kaufmann, 1992.

[14] J. Huang. MUP: A minimal unsatisfiability prover. In *Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, pages 432–437, Shanghai, China, 2005.

[15] H. Kautz, B. Selman, and D. McAllester. Walksat in the SAT 2004 competition. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, 2004.

[16] M. Liffiton and K. Sakallah. On finding all minimally unsatisfiable subformulas. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 173–186, 2005.

[17] I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, 2004.

[18] B. Mazure, L. Saïs, and É. Grégoire. A powerful heuristic to locate inconsistent kernels in knowledge-based systems. In *International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU'96)*, pages 1265–1269, Granada (Spain), 1996.

[19] B. Mazure, L. Saïs, and É. Grégoire. Boosting complete techniques thanks to local search. *Annals of Mathematics and Artificial Intelligence*, 22:319–322, 1998.

[20] Y. Oh, M. Mneimneh, Z. Andraus, K. Sakallah, and I. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *Design Automation Conference (DAC'04)*, pages 518–523, 2004.

[21] C. Papadimitriou and D. Wolfe. The complexity of facets resolved. *Journal of Computer and System Sciences*, 37(1):2–13, 1988.

[22] SATLIB. Benchmarks on SAT. http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html.

[23] L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, Portofino (Italy), 2003.

Annex: proofs

Proof of property 1 By definition, strict inconsistent covers contain MUSes with empty intersections. Since at least one clause per MUS is falsified under any interpretation I, at least $|IC|$ clauses are thus falsified under I. □

Proof of property 2 If C is critical then for each literal l of C, $\exists\ C'$ s.t. C' is once-satisfied w.r.t. I and $\bar{l}$ belongs to C'. C is falsified under I, thus l is $false$ under I and $\bar{l}$ is $true$ under I. $\bar{l}$ is the only satisfied literal of C', accordingly if the value of l is reversed then C' becomes falsified. □

Proof of property 3 Any falsified clause under I belongs to a MUS because I is optimal w.r.t. the number of satisfied clauses and at least one clause of each MUS cannot be satisfied. The fact that any falsified clause under I is critical is proved thanks to Property 5 since I is a global mimimum. I is optimal w.r.t. the number of satisfied clauses, thus at most one clause per MUS is falsified. Also, if one flip allows us to satisfy one of theses clauses, another clause of the MUS becomes falsified. Accordingly, at least one once-satisfied clause linked to a falsified clause under I belongs to a MUS of Σ. □

Proof of property 4 Let Γ be a MUS and C be a clause s.t. $C \in \Gamma$. By definition of a MUS, $\Gamma \setminus C$ is satisfiable. Let M be a model of $\Gamma \setminus C$. Let us prove that C is critical w.r.t. M. First, C is falsified. Indeed, if C is not falsified then Γ exhibits a model M. This is impossible because Γ is a MUS. Second, C is critical. Indeed, if any variable occurring in C is flipped w.r.t. M, then at least one clause of Γ becomes falsified since Γ is unsatisfiable. That means that this newly falsified clause was once-satisfied and linked to C. Accordingly, C is critical w.r.t. M. □

Proof of property 5 If a variable occurring in a falsified clause w.r.t. a minimum is flipped, then this clause is satisfied and at least one previously satisfied clause becomes unsatisfied. That means that this new unsatisfied clause was once-satisfied. Accordingly, the initial falsified clause was critical. □

Ario: A Linear Integer Arithmetic Logic Solver

Hossein M. Sheini
Electrical Engineering and Computer Science Dept.
University of Michigan, Ann Arbor, MI 48109, USA
Email: hsheini@umich.edu

Karem A. Sakallah
Electrical Engineering and Computer Science Dept.
University of Michigan, Ann Arbor, MI 48109, USA
Email: karem@umich.edu

***Abstract*— In this paper we describe our solver for systems of linear integer arithmetic logic. Such systems are commonly used in design verification applications and are classified under Satisfiability Modulo Theories (SMT) problems. Recognizing the fact that in many such applications the majority of atoms are equalities or integer unit-two-variable inequalities (UTVPIs), we present a framework that integrates specialized theory solvers for those atoms within a SAT solver. The unique feature of our strategy is its simultaneous adoption of both a congruence-closure equality solver and a transitive-closure UTVPI solver to find a satisfiable set of those atoms. A full-scale ILP solver is then utilized to check the consistency of all integer constraints within the solution. Other notable features of our solver include its combined deduction and learning schemes that collectively make our solver distinct among similar solvers.**

I. Introduction

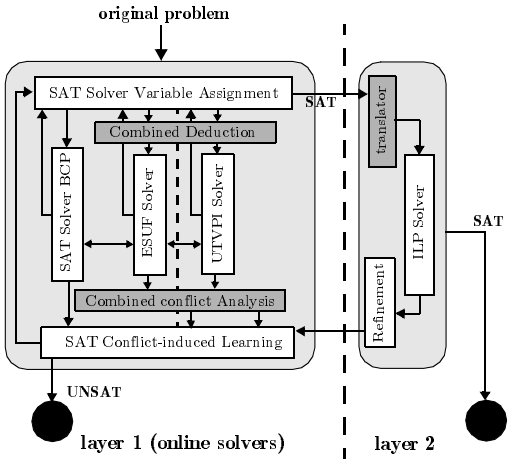
In the past few years there has been a noticeable surge in the introduction and application of various logics to model systems of integer constraints. Different design verification problems are routinely cast in terms of logical problems whose atoms are constraints over integer variables.

In this paper we briefly describe Ario, a solver for checking the satisfiability of quantifier-free formulas in linear integer arithmetic logic[1]. Ario essentially is able to deal with two main logics: i) equality logic with successors and uninterpreted functions (ESUF), whose atoms are in the form of $t_i = t_j + c$ where $c \in \mathbb{Z}$ and t_i, t_j are terms, that are recursively defined to be either integer variables or applications of uninterpreted functions over terms, and ii) linear integer arithmetic logic (LIA) whose atoms are linear constraints over integer variables. We assume that ESUF and LIA atoms within the input problem do not share variables; i.e. an equality over two integer variables, $t_i = t_j + c$, is an ESUF atom only if t_i and t_j are not present in any inequalities; otherwise it is replaced with a conjunction of two LIA atoms: $t_i - t_j \leq c$ and $t_j - t_i \leq -c$. We additionally categorize LIA atoms into two types, each to be treated differently: i) Unit-Two-Variable-Per-Inequality (UTVPI) atoms in the form of $a_i x_i + a_j x_j \leq b$ where x_i, x_j are integer variables, $a_i, a_j \in \{0, \pm 1\}$ and $b \in \mathbb{Z}$, and ii) non-UTVPI atoms in the form of $\sum_{i=1}^{n} a_i x_i \leq b$ where $a_i, b \in \mathbb{Z}$ and x_i's are integer variables.

Ario adopts a generic CNF SAT solver to reason about and analyze the logical structure of the problem. Following the DPLL framework, the SAT solver incrementally builds a satisfiable set of atoms and utilizes specialized *theory solvers* to maintain the consistencies of the conjunctions of various *theory atoms* within that solution. Upon activating ESUF or UTVPI atoms, the consistencies of those atoms are incrementally checked using, respectively, a congruence-closure or a transitive-closure procedure. The overall consistency of all LIA atoms is established after/if a satisfiable solution to the rest of the problem is built.

[1]For complete details on the overall algorithm of Ario and its different techniques the reader is referred to [1] and [2].

Fig. 1. The organization of solving algorithms within the Ario solver



Related Work: The idea of integrating different theory solvers within a propositional SAT solver was first suggested in [3] and then further improved in [4], [5], [6], [2]. Two of the most common integration strategies employed in these solvers include, i) the *layered approach* [5], based on invoking theory solvers in the order of their solving capabilities, and ii) the online [4] or DPLL(T) [6] approach, where a combined DPLL reasoning and learning procedure is applied to all atoms and the consistency of the theory atoms are maintained throughout the SAT search. The latter is essentially a form of "early pruning" as introduced in [3].

II. The Solver Architecture

The overall architecture of Ario is demonstrated in Figure 1. In this section we describe each of Ario's theory solvers, and the framework for utilizing these solvers within SAT.

A. Theory Solvers

The following three solving procedures are utilized in Ario, each capable to decide the consistency of a conjunction of specific theory atoms. Considering that the ESUF and UTVPI

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

solvers are integrated within the SAT solver, their associated algorithms are both incremental and backtrackable.

1) Solving Equalities with Successors and Uninterpreted Functions (ESUF): This solver initially replaces all applications of the successor functions by simple addition of their arguments with integer constants and subsequently adopts a reimplementation of the congruence-closure algorithm of [7] to solve the conjunction of ESUF atoms.

2) Solving UTVPI Constraints: Upon activating a UTVPI atom by the SAT solver, that atom is added to a transitively-closed and tightened set of constraints as in [8]. A conflict is detected if a constraint of the form $0 \leq -1$ is implied.

3) Solving Integer Linear Constraints: This solver adopts a generic Simplex/Branch-and-Bound methods to establish the satisfiability of the integer constraints. Activated non-UTVPI atoms together with only those UTVPI atoms that share variables with them are solved with this solver [2].

B. Integration within SAT

In Ario we implemented a *hybrid approach* [1] to integrate theory solvers within SAT. In this framework, similar to early pruning approach of [3], those theory solvers that can efficiently adapt to the DPLL SAT procedure are integrated online and the rest are utilized in separate layers and are only applied to complete SAT solutions. More specifically, the ESUF and UTVPI solvers are integrated within SAT while non-UTVPI constraints are solved in an offline layer (as demonstrated in Figure 1 and further described in [2]). Our hybrid method enables the solver to efficiently process ESUF and UTVPI atoms and only check the consistency of hard non-UTVPI integer constraints when it is absolutely necessary, i.e. when a satisfiable assignment to Boolean, ESUF and UTVPI atoms is found. Further enhancements due to adopting our combined deduction and learning schemes are as follows.

1) Combined Deduction Scheme: Ario utilizes an inter-logic deduction scheme [1] outside the theory solvers that builds an implication graph taking into account all types of atoms. Implications in this graph are both due to unit-clause-propagation of the SAT solver and the linear combination of integer constraints. In this scheme all possible non-negative linear combinations of equality and UTVPI constraints are generated and implied. Non-UTVPI constraints are only combined with equality or UTVPI constraints if a variable could be eliminated and they are not combined with other non-UTVPI constraints. These combinations could generate new atoms that are not present in the original formula to be added to their respective theory solvers on the fly. This method enables the solver to reason about theory atoms outside their specific theory solvers and helps each solver within our framework to prune infeasible solutions due to other types of theory atoms. For further details, the reader is referred to [1].

2) Combined Learning Scheme: By analyzing the combined deduction scheme at each conflict, this method learns clauses in terms of different types of theory atoms. For instance, if a conflict is detected among non-UTVPI atoms, the combinations of non-UTVPI and UTVPI atoms are considered to learn a clause in terms of only UTVPI atoms. This clause can then be used in the online search to reduce the number of calls to the costly ILP solver. This is further explained in [2].

III. The Solver Performance

Table I demonstrates a comparison on the number of instances in various benchmark suites that Ario and each of its competitors could solve within 600 seconds. For this experiment, we used the benchmarks from SMT Library [9] and compared Ario to MathSAT v3.3.1 (MSAT) [5], YICES v0.1 [10] and BarceLogicTools (BCLT) [6]. All experiments were conducted on an AMD Opteron 2.2GHz (8GB RAM) machine.

TABLE I

Comparing solving methods within the time limit of 600 sec.

logic	benchmark suite	# of inst's	Number of instances solved			
			Ario	MSAT	YICES	BCLT
QF-LIA	wisas	108	**103**	62	97	NA
QF-LIA	CIRC	51	**33**	30	27	NA
QF-LIA	fischer-fair	121	97	110	**112**	NA
QF-UFIDL	uclid	47	44	38	41	**45**
QF-UFIDL	pete	233	**233**	82	147	**233**
QF-UF	EQs	152	109	99	106	**115**

IV. Summary

In this paper we presented our Ario SAT solver for linear integer arithmetic logic. Ario adopts a hybrid integration approach, i.e. it applies the online integration strategy to check the consistency of ESUF and UTVPI atoms within SAT and the layered approach for non-UTVPI integer constraints. This approach for categorizing integer constraints together with our framework specifically adapt to applications in design verification where the majority of integer constraints are either equalities or UTVPIs[2].

References

[1] H. M. Sheini and K. A. Sakallah, "A progressive simplifier for satisfiability modulo theories." in *SAT*, 2006, pp. 184–197.

[2] ——, "A scalable method for solving satisfiability of integer linear arithmetic logic." in *SAT*, 2005, pp. 241–256.

[3] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani, "A SAT based approach for solving formulas over boolean and linear mathematical propositions," in *CADE-18*, 2002, pp. 195–210.

[4] S. Berezin, V. Ganesh, and D. L. Dill, "An online proof-producing decision procedure for mixed-integer linear arithmetic," in *TACAS*, 2003, pp. 521–536.

[5] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani, in *TACAS*, 2005, pp. 317–333.

[6] R. Nieuwenhuis and A. Oliveras, "DPLL(T) with exhaustive theory propagation and its application to difference logic." in *CAV*, 2005, pp. 321–334.

[7] ——, "Proof-Producing Congruence Closure," in *RTA*, 2005, pp. 453–468.

[8] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap, "Beyond finite domains," in *Workshop on Principles and Practice of Constraint Programming*, 1994, pp. 86–94.

[9] "The Satisfiability Modulo Theories Library." [Online]. Available: http://combination.cs.uiowa.edu/smtlib/

[10] L. de Moura, "Yices," 2005. [Online]. Available: http://fm.csl.sri.com/yices/

[2]This work was funded in part by the National Science Foundation under ITR grant No. 0205288.

Understanding the Dynamic Behaviour of Modern DPLL SAT Solvers through Visual Analysis

Cameron Brien, Sharad Malik
Department of Electrical Engineering
Princeton University
{cbrien,sharad}@princeton.edu

Abstract

Despite the many recent improvements in the speed and robustness of DPLL-based SAT solvers, we still lack a thorough understanding of the working mechanisms and dynamic behaviour of these solvers at run-time. In this paper, we present TIGERDISP, a tool designed to allow researchers to visualize the dynamic behaviour of modern DPLL solvers in terms of time-dependent metrics such as decision depth, implications and learned conflict clauses. It is our belief that inferences about dynamic behaviour can be drawn more easily by visual analysis than by purely aggregate post-execution metrics such as total number of decisions/implications/conflicts. These inferences can then be validated through detailed quantitative analysis on larger sets of data. To this end, we have used TIGERDISP with the HAIFASAT and MINISAT solvers and have generated a few specific inferences about their relatively efficient and inefficient solving runs. We have then tested one of these inferences through quantitative analysis on a larger data set and have presented our findings in this paper. An important application of TIGERDISP would be in the development of a solver that employs adaptive algorithms. This is an area that has intrigued researchers in the past, but has not seen significant results for lack of a clear understanding as to what constitutes good progress during the run of a SAT solver. With better knowledge of dynamic behaviour, it is conceivable that an adaptive solver could be designed such that it switches between several competitive heuristics at run-time based on a quantitative analysis of its own dynamic behaviour.

I. Introduction

Despite being one of the most exhaustively studied problems in combinatorial search, there continue to be significant advances in the field of Boolean Satisfiability. Even over the past five years, we have seen improvements of orders of magnitude in the speed and robustness of SAT solvers. Nonetheless, a thorough understanding of the working mechanisms and behaviours of solvers at run-time is still lacking. Our knowledge of the measure of the SAT solution process is limited to post-execution analysis of net metrics. As such, today's SAT solver development roughly follows a three-stage process: an algorithm is proposed, it is implemented in code and, finally, it is tested and tuned against a mixture of industry-generated and randomized inputs.

In this paper, we discuss our first attempts at gaining some insight into the SAT solution process for DPLL-based solvers. In particular, we present TIGERDISP, an *interactive* display tool for the visual analysis of SAT solver performance. Our tool meaningfully displays run-time metrics gathered at every iteration of the solution process, allowing users to zoom in and examine a solver's dynamic behaviour *at any granularity*.

It is our belief that there is much to be gained from a better understanding of the dynamic behaviour of modern solvers. First off, if we can quantify relatively efficient and inefficient behaviour, we can use this insight directly to develop faster solver heuristics. Perhaps more exciting, however, is that if we can do this quantitative analysis efficiently, we can develop adaptive solvers that dynamically switch between different heuristics on the fly for increased robustness and reduced mean execution time.

II. SAT Solvers

In this paper, we concern ourselves with modern Davis-Putnam-Logemann-Loveland variants, specifically those that make use of learned conflict clauses [1]. Although our display tool was designed to be versatile, it has special features for the analysis of such solvers. While developing and testing TIGERDISP, we have made use of the HAIFASAT and MINISAT solvers – they were among the strongest competitors in the SAT 2005 COMPETITION [2] [3] [4].

III. Metrics of Importance

It is our hope that a number of different SAT research groups may use TIGERDISP to analyze the performance of their solvers and make improvements to their heuristics. As such, our tool has been designed with generality in mind and can be used to track a variety of run-time metrics and compare them against those of any number of other solvers.

In these early stages of our research, we have chosen to use TIGERDISP to study three of the most basic metrics of any modern DPLL solver: the decision depth, total number of implications and the length and quantity of learned conflict clauses. We have also chosen to track the number of cache misses as it is a good measure of the general performance of any piece of modern software.

The magnitude of the depth is of interest because large depths maintained for extended periods of time may indicate that a solver is "trapped" in a search space with no solution. Since implications are forced decisions, we monitor the implication depth for similar reasons. Learned conflict clauses are also of great interest: they allow solvers to avoid previous mistakes but, at the same time, scanning through these clauses takes time. To this end, our tool records the quantity of learned clauses in memory at each iteration – the point at which the solver makes a new, unforced decision – and provides a dynamic histogram of their sizes.

The tool takes metric dumps from different solvers and plots the data sets against each other in a meaningful way. The dumps are flat files generated by inserting simple metric outputting code throughout the original solver source. In our trials, for example, a dump consists of whitespace-delimited flat files containing a listing of the decision depth, total number of implications, learned clause lengths and the number of cache misses for every iteration of the solving process. Solver execution with our metric monitoring averaged less than 0.5% slower than execution without the additional code.

IV. TigerDisp

The most recent version of our display tool is written in Java and works across platforms. In laying out our tool, we were in

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

a Harvard student's display project, "Idaho Tree Rings" [5]. Given the object-oriented nature of our code, the display tool allows one to compare any number of metrics for any number of different solvers. In the case of a two-solver comparison, for example, one solver's metrics are displayed on the left and the corresponding metrics of the other solver are displayed on the right. The vertical scale of any pair of metrics from the two solvers is relative in order to allow for meaningful comparison. The horizontal (time) scale is also the same; this is achieved by simply cropping the metrics of the slower solver that continue past the point of the first solution.

TIGERDISP is *not* a static graph plotter; rather, it is interactive tool that allows users to dynamically track a variety of different solver metrics. The tool's waveform functionality allows the user to plot metrics against iteration (decision) number or real time, and these plots can be viewed at any granularity. The tool also allows the user to display metrics in a histogram. The histogram changes as the mouse is used to scan along the solution process. This allows the user to view metrics in a histogram plot and monitor the changes to the plot against iteration number or time. One can also use the tool to zoom in and out and to examine the actual values of the metrics at any point along the waveforms or histograms. The dynamic nature of TIGERDISP can at best be weakly conveyed through the static pictures available in a paper. Instead, we prefer that readers view the detailed demonstration movie at: `http://www.princeton.edu/~chaff/tigerdisp`. TIGERDISP is available for download from the same location.

V. INITIAL OBSERVATIONS

Having developed TIGERDISP, we have entered the early analysis phase of our research. To date, we have used the tool to analyze a number of runs of the HAIFASAT and MINISAT solvers and have generated a few specific inferences about their relatively efficient and inefficient solving runs. While significantly more time is needed to investigate these inferences, we have pursued one of our findings in depth for the sake of illustration in this paper. Here, we take an inference based on visual cues from TIGERDISP and perform numerical analysis on a larger data set to test and possibly refine the inference.

In unsatisfiable instances, we have observed that the HAIFASAT solver consistently has very deep decision depths in the early stages of the solution process and shallow depths for the remainder of the process. In contrast, MINISAT maintains very shallow decision depths for the entire solution process.

A solver's decision depth waveform is indicative of its particular approach to localizing its effort within a search space. As such, we have hypothesized that these waveforms could be linked to the relative efficiency of the two solvers. Through numerical analysis, we have examined the average decision depths of the two solvers on the final 80% of the duration of a run, for unsatisfiable instances. (The exclusion of the first 20% allows us to ignore the initial spike observed in HAIFASAT runs.)

We have not determined a clear relationship between these relative averages and the relative performance of the two solvers; however, we did make one interesting finding. In all unsatisfiable instances, in the final 80% of the solving process HAIFASAT maintained a greater average decision depth than MINISAT. The mean difference of the average depths was a factor of approximately 2. So, while our very early investigation into the relationship between average decision depth and relative efficiency has not yielded striking results, we have taken a visual inference and used it as the cue for numerical analysis. This numerical analysis, in turn, has exposed an interesting and consistent difference between the two solvers' average decision depths.

VI. NEXT STEPS

Our next steps involve running visual analysis with TIGERDISP on a greater number of metric dumps from a greater number of solvers and generating more specific inferences; these inferences can then be further investigated through numerical analysis. In addition to a larger set of dumps, we would like to monitor a broader spectrum of metrics. It seems that there would also be a significant value in using the tool to analyze different parameterizations of the same solver as this could demonstrate a new and more efficient method for developers to tune their solvers' parameters.

An important application of TIGERDISP may be in the development of a solver that employs adaptive algorithms. This is an area that has intrigued researchers in the past, but has not seen significant results for lack of a clear understanding as to what constitutes good progress during the run of a SAT solver [6]. With a better knowledge of dynamic behaviour, it is conceivable that an adaptive solver could be designed such that it switches between several competitive heuristics at run-time based on a quantitative analysis of its own dynamic behaviour. Another possible design would involve running a number of solvers in parallel threads and weighting the threads' execution priority based on an analysis of their relative dynamic efficiency.

VII. CONCLUSIONS

This paper lays the groundwork for a new approach to SAT solver analysis. Here, we present TIGERDISP, a tool designed to allow researchers to visualize the dynamic behaviour of modern DPLL solvers in terms of time-dependent metrics such as decision depth, implications and learned clauses. It is our belief that inferences about dynamic behaviour can be drawn more easily through visual analysis than through purely numerical methods. These inferences can then be validated through detailed quantitative analysis on larger sets of data.

Our very early visual inferences have demonstrated interesting and consistent differences between the HAIFASAT and MINISAT solvers. Additionally, we have demonstrated that results from TIGERDISP can be used to focus our quantitative efforts.

APPENDIX

Our observations are based on runs of HAIFASAT and MINISAT with the SAT 2002 COMPETITION benchmarks: Aloul's BART, HOMER and LISA, Biere's CMPADD and DINPHIL, Dellacherie's IP, Goldberg's BMC2, Van Gelder's ROPEBENCH and Velev's FVP and SSS [7].

REFERENCES

[1] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, July 1962.

[2] O. Strichman and R. Gershman, "HAIFASAT: A new robust SAT solver," *Haifa Verification Conference*, 2005.

[3] N. Eén and N. Sörensson, "MINISAT - a SAT solver with conflict-clause minimization," *The International Conferences on Theory and Applications of Satisfiability Testing*, 2005.

[4] "SAT competitions," Website, Accessed 28 February 2006, http://www.satcompetition.org.

[5] R. Hohenstein, "Assignment 11b: Built with processing," Website, Accessed 10 November 2005, http://ves61r.net/ray/11b/tree/applet.

[6] O. Shacham and K. Yorav, "Adaptive application of SAT solving techniques, IBM Labs in Haifa," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 1, 2006.

[7] "The SAT 2002 competition," Website, Accessed 1 March 2006, http://www.satcompetition.org/2002/onlinereport.pdf.

Over-Approximating Boolean Programs with Unbounded Thread Creation

Byron Cook
Microsoft Research
Cambridge
Email: bycook@microsoft.com

Daniel Kroening
Computer Systems Institute
ETH Zurich
Email: daniel.kroening@inf.ethz.ch

Natasha Sharygina
Computer Science Department
University of Lugano
Email: natasha.sharygina@unisi.ch

***Abstract*— This paper describes a symbolic algorithm for over-approximating reachability in Boolean programs with unbounded thread creation. The fix-point is detected by projecting the state of the threads to the globally visible parts, which are finite. Our algorithm models recursion by over-approximating the call stack that contains the return locations of recursive function calls, as reachability is undecidable in this case. The algorithm may obtain spurious counterexamples, which are removed iteratively by means of an abstraction refinement loop. Experiments show that the symbolic algorithm for unbounded thread creation scales to large abstract models.**

I. Introduction

All scalable symbolic model checkers for software are currently based on counterexample-guided abstraction refinement (CEGAR) (e.g., BLAST [1], SLAM [2], MAGIC [3], SATABS [4], DIVER [5]). To date, none of these model checkers supports unbounded thread creation together with shared memory cross-thread communication. This gap is not due to lack of need: much of the software that these tools are used to verify are actually shared memory concurrent programs with unbounded thread creation. Static Driver Verifier (SDV) [2], for example, is used to verify Windows device drivers—which are tremendously concurrent pieces of software. SDV's analysis is unsound because it ignores the side-effects caused by other threads.

The cause for this gap between the software model checkers and the software that they are intended to verify is a technical one: CEGAR is effective only if the underlying reachability procedure is guaranteed to terminate—and terminate quickly. When unbounded thread creation is added into the mixture, today's reachability engines often do not terminate.

We address this problem with a new symbolic model checker for *Boolean programs* (the most common form of abstractions used within CEGAR-based tools for software) that supports unbounded thread creation while guaranteeing termination. What we lose is precision—the Boolean program checker may now return counterexamples that are spurious within the abstraction itself. The experimental results show that this is not a practical problem: the CEGAR refinement mechanism can be adapted to remove these false counterexamples as well as the counterexamples that are spurious only in the unabstracted software. Furthermore, the experimental results demonstrate that the algorithm scales in practice to large concurrent programs.

The contribution of this paper is contained in Sections III and IV, namely an algorithm for reachability analysis of programs with unbounded thread creation in Section III and their symbolic simulation in Section IV. Experimental results are discussed in Section V.

Related Work

Formal verification of multi-threaded programs is an area of active research; see [6] for an excellent survey. The development of static analysis tools for such programs is complicated due to the fact that reachability for interprocedural programs (that is, for programs that contain both communication and data-flow structures) is undecidable [7].

Pushdown automata have been used as tools for analyzing sequential programs with (recursive) procedures [8]. The expressive power of pushdown systems is equivalent to that of sequential programs with (possibly recursive) procedures where all variables have a finite data type. MOPED [9] and BEBOP [10], for example, are BDD-based symbolic model checkers for this class of language. There has also been work on pushdown automata with multiple stacks, e.g., see [11]. As reachability is undecidable in this case, the existing implementations are not fully automated.

Unsound approaches have also proved successful in finding bugs in concurrent programs. For example, Qadeer & Rehof [12] note that many bugs can be found when the analysis is limited to execution traces with only a small set of context-switches. This analysis supports recursive programs. Our approach complements these techniques because, while they are unsound, they are able to analyze a larger set of programs.

The class of programs considered in this paper can be viewed as an instance of a *parameterized system*, i.e., a system with a number of identical processes (threads in our case). Many approaches to this problem have been developed over the years, including the use of symbolic automata-based techniques, network invariants, predicate abstraction or system symmetry (see an excellent overview in [13]). Methods that are most closely related to our work are based on abstraction (for example, an extension of Murϕ uses abstraction for replicated identical components [14]). In contrast to our approach, many of these methods are only partially automated, requiring at least some human ingenuity to construct a process invariant

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

or a closure process (for example, the TLPVS tool [15] is based on manual theorem proving).

Henzinger et al. use predicate abstraction in order to construct environment models from threads [16]. When combined with a counter abstraction, an unbounded number of threads can be supported. Flanagan and Qadeer propose to use the idea of thread-states in order to obtain environment models for *loosely-coupled* multi-threaded programs [17]. In contrast to their algorithm, we address the spurious behavior introduced by this over-approximation by (safely) restricting the thread-states that are passed, and by an automatic refinement procedure.

One can also model concurrent Boolean programs as a set of rewriting rules and use rewriting techniques to prove safety. For example, [18] computes abstractions of program paths using the least solutions of a system of path language constraints. At this time it is not clear how our work compares to these techniques. One disadvantage of the term rewriting approach is that it requires translating programs written in general purpose languages into the term models. There are no translation tools reported yet.

A number of tools for analysis of multi-threaded Java programs is available. While some of the tools compute abstract models automatically, most perform only explicit state space exploration. Representative examples of model checkers for Java are [19] and JPF [20]. Yahav reports an implementation of a Model Checker for Java with an unbounded number of threads using three-valued logic [21]. Similarly to our approach, an over-approximation is computed.

The reachability of concurrent programs with a restricted form of recursion is shown to be decidable and implemented in ZING [22]. Here, recursive functions are partitioned into *atomic transactions*, which are only allowed to modify local variables. ZING, however, suffers from scalability problems since its approach flattens concurrent programs to sequential programs to handle recursive procedures. BEACON [23] (an explicit-state model checker for concurrent Boolean programs) has similar scalability problems. Additionally, CEGAR-based tools produce abstractions that make non-trivial use of underspecified values. For this reason, explicit-state model checkers perform poorly when used as the reachability procedure within a CEGAR loop.

II. BOOLEAN PROGRAMS

A. Syntax

The syntax of the control flow statements is derived from C, and can be found in [10]. The syntax for expressions permits the usual Boolean operators, and the following two extensions: 1) non-deterministic choice, and 2) next-state variables.

$$
\begin{array}{rcl}
\mathit{expression} & : & \mathit{expression}\ \text{`}\vee\text{'}\ \mathit{expression} \\
& | & \mathit{expression}\ \text{`}\wedge\text{'}\ \mathit{expression} \\
& | & \text{`}\neg\text{'}\mathit{expression} \\
& | & \mathit{atom} \\
\mathit{atom} & : & \mathit{Identifier} \mid \mathit{Identifier}\ \text{`}'\text{'} \mid\ \text{`}\star_j\text{'}
\end{array}
$$

The stars denote non-deterministic choice symbols. If multiple non-deterministic choices are to be used in one expression, we number them $\star_1, \star_2, \ldots$[1]. If an identifier is followed by a prime, the identifier is to be evaluated in the next state.[2]

B. Formal Semantics

We extend the semantics of Boolean Programs [10] to permit unbounded thread creation. Let V_g denote the set of global variables. For the sake of simplicity, we assume that all threads have the same set of local variables V_l and the same program code, i.e., there is only one set of program locations L. We denote the program by P. A program with threads that have different code can easily be transformed into a program with identical threads. We denote the set of variables by $V = V_g \dot\cup V_l$. We assume that a subset $\mathcal{L} \subseteq V_l$ of the local variables is used for locking exclusively.[3]

Definition 1 (Explicit State): An *explicit state* η of a Boolean program is a triple (n, pc, Ω), where $n \in \mathbb{N}$ is the number of threads, $pc : \{1, \ldots, n\} \mapsto L$ is the vector of program locations, $\Omega : (\{1, \ldots, n\} \times V_l) \cup V_g \mapsto \mathbb{B}$ is the valuation of the program variables. We denote the set of explicit states by S.

We denote the projection of a state η to the number of running threads in that state by $\eta.n$, the projection from a state to the values of the program counters by $\eta.pc$, and so on. The value of the program counter of thread $t \in \{1, \ldots, n\}$ is denoted by $\eta.pc(t)$, the value of the local variable $v \in V_l$ of thread t is denoted by $\eta.\Omega(t, v)$.

Definition 2 (Thread State): The tuple (PC, Ω) with $PC \in L$ and $\Omega : V \longrightarrow \mathbb{B}$ is called a *thread state*. It is a valuation of the program counter, the local variables of a particular thread, and the global shared variables. We use $\tilde{S}$ to denote the set of thread states.

Thus, the thread state is the set of values that is *visible* to a thread.

Definition 3 (μ_t): The *thread state projection function* $\mu_t : S \longrightarrow \tilde{S}$ takes a state η of the full state space and maps it to the state visible to thread $t \in \{1, \ldots, \eta.n\}$.

$$
\begin{array}{rcl}
\mu_t(\eta).PC & := & \eta.pc(t) \\
\mu_t(\eta).\Omega(v) & := & \left\{ \begin{array}{lcl} \eta.\Omega(v) & : & v \in V_g \\ \eta.\Omega(t, v) & : & v \in V_l \end{array} \right.
\end{array}
$$

Given a thread state $\tilde{\eta} \in \tilde{S}$ and an expression e over the variables V, we use $[\![e\,,\, \tilde{\eta}]\!]$ to denote the evaluation of e by a thread in state $\tilde{\eta}$. Let e, e_1, and e_2 denote expressions, and $v \in V$ be a variable. Formally, $[\![e\,,\, \tilde{\eta}]\!]$ is defined recursively as follows:

$$
\begin{array}{rcl}
[\![e_1 \vee e_2\,,\, \tilde{\eta}]\!] & := & [\![e_1\,,\, \tilde{\eta}]\!] \vee [\![e_2\,,\, \tilde{\eta}]\!] \\
[\![\neg e\,,\, \tilde{\eta}]\!] & := & \neg [\![e\,,\, \tilde{\eta}]\!] \\
[\![v\,,\, \tilde{\eta}]\!] & := & \tilde{\eta}.\Omega(v)
\end{array}
$$

[1] The `schoose` non-deterministic choice operator implemented by BEBOP can be transformed into an expression that uses $\star$.

[2] Tools such as BEBOP expect the prime *before* the identifier.

[3] We use local variables instead of global variables for locking in order to be able to identify the individual thread that holds a lock.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

The next-state identifiers (primed identifiers) refer to the next thread state $\tilde{\zeta} \in \tilde{S}$. The semantics of expressions containing such primed identifiers is defined using the evaluation function $[\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}]\!]$. The definition of $[\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}]\!]$ is identical to the definition of $[\![e\,,\,\tilde{\eta}]\!]$ above, unless e is a primed identifier:

$$[\![v'\,,\,\tilde{\eta}\,,\,\tilde{\zeta}]\!] \quad := \quad \tilde{\zeta}.\Omega(v)$$

The semantics of expressions containing non-deterministic choice symbols is given by $[\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]$, where ι denotes the valuation of the $\star$ symbols. The definition is identical to the definition above, unless e is $\star_j$ for some j:

$$[\![\star_j\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!] \quad := \quad \iota_j$$

For any function $f : D \to R$ and any $d \in D, r \in R$, we define $f[d/r] : D \to R$ as follows:

$$f[d/r](x) = \begin{cases} r & : \quad d = x \\ f(x) & : \quad \text{otherwise} \end{cases}$$

As a shorthand, we write $\tilde{\eta} \stackrel{G}{=} \tilde{\zeta}$ iff the values of the global, i.e., shared variables in $\tilde{\eta}$ and $\tilde{\zeta}$ are equal, i.e., $\forall g \in V_g.[\![g\,,\,\tilde{\eta}]\!] = [\![g\,,\,\tilde{\zeta}]\!]$. Similarly, we write $\tilde{\eta} \stackrel{L}{=} \tilde{\zeta}$ iff the values of the local variables in $\tilde{\eta}$ and $\tilde{\zeta}$ are equal.

Execution Semantics: We use $\tilde{\eta} \longrightarrow \tilde{\zeta}$ to denote the fact that a transition from state $\tilde{\eta}$ is made to $\tilde{\zeta}$ by executing the statement $\tilde{\eta}.PC$. The relation $\tilde{\eta} \longrightarrow \tilde{\zeta}$ is defined by a case-split on this instruction. The conditions for each case are shown in Table I. The description of the semantics of the `skip`, `goto`, `assume`, and constrained assignment statements are identical to the description found in [24]. The definitions of `lock` and `unlock` are straight-forward. Note that `lock` and `unlock` are special cases of a constrained assignment.

We write $l(\tilde{\eta}) \subseteq \mathcal{L} := \{l \in \mathcal{L} \,|\, \tilde{\eta}.\Omega(l)\}$ for the set of locks that are held in state $\tilde{\eta}$.

The semantics of the concurrent program is defined as follows: Assume the scheduler picks a thread $t \in \{1, \ldots, \eta.n\}$ to execute in state η. We use $\eta \longrightarrow_t \zeta$ to denote the fact that a transition from state η is made to ζ by executing one statement of thread t. The statement that is executed is $P(\eta.pc(t))$. The relation $\eta \longrightarrow_t \zeta$ is defined by a case-split on this instruction. For all instructions but `start_thread`, we require that

- the number of threads does not change, i.e., $\zeta.n = \eta.n$,
- thread t makes a transition, i.e., $\mu_t(\eta) \longrightarrow \mu_t(\zeta)$,
- and the values of local variables and the program counters of the other threads $j \neq t$ remain unchanged, i.e., $\mu_j(\eta) \stackrel{L}{=} \mu_j(\zeta)$ and $\zeta.pc(j) = \eta.pc(j)$,
- locks are held exclusively, i.e., $l(\mu_u(\zeta)) \cap l(\mu_v(\zeta)) = \emptyset$ for all $u \neq v$.

If $P(\eta.pc(t))$ is `start_thread` θ, we require that

- the number of threads increases by one, i.e., $\zeta.n = \eta.n + 1$,
- the program counter of the new thread is θ, and the program counter of thread t is $\eta.pc(t)+1$, i.e., $\zeta.pc(\zeta.n) = \theta$ and $\zeta.pc(t) = \eta.pc(t) + 1$,
- thread t makes a transition into both changed states, i.e., $\mu_t(\eta) \longrightarrow \mu_t(\zeta)$ and $\mu_t(\eta) \longrightarrow \mu_{\zeta.n}(\zeta)$, and
- the values of the local variables of the other threads $j \neq t$ and $j \neq \zeta.n$ remain unchanged, i.e., $\mu_j(\eta) \stackrel{L}{=} \mu_j(\zeta)$ and $\zeta.pc(j) = \eta.pc(j)$.

Syntactic sugar such as `if` or `while` can be easily transformed using `goto` and `assume`, as described in [24]. For now, we assume that function calls can be inlined. We extend our algorithm to support unbounded recursion in Section IV-B.

Finally, we write $\eta \longrightarrow \zeta$ if there exists a thread $t \in \{1, \ldots, \eta.n\}$ such that $\eta \longrightarrow_t \zeta$. In this case, we say that there is a transition from η to ζ, or that ζ is reachable from η with one transition. Let $S^0 \subseteq S$ denote the set of initial states, and let $S^i \subseteq S$ with $i \in \mathbb{N}$ denote the set of states reachable in i or less transitions. The set of all reachable states is S^∞. The property we check is reachability of states with particular program locations.

III. Over-Approximation and Refinement

A. Over-approximating S^∞

Finite-state model checking algorithms are based on fix-point detection, that is, the model checker compares the new set of states computed using the transition relation with the states explored so far. The algorithm iterates until no new states are discovered.

This basic idea can be applied to programs with unbounded thread creation as well. For example, SPIN [25] permits dynamic creation of new threads by means of Promela's `run` statement. However, SPIN assumes that the program only creates a finite number of threads. If the thread creation is not actually bounded, the state enumeration of SPIN never terminates.

We propose an algorithm that does not restrict thread creation to a finite number, i.e., we permit an infinite set S^∞ while still guaranteeing termination. The classical fix-point detection algorithm is not readily applicable for this case.

Definition 4 (μ_*): Let $\mu_*(\eta)$ denote the set of the thread states $\mu_t(\eta)$ for any thread t. Let $S' \subseteq S$ be a set of states. The *thread-visible states* are the states in S' projected to the thread states of all threads.

$$\mu_*(\eta) := \textstyle\bigcup_{t \in \{1,\ldots,\eta.n\}} \{\mu_t(\eta)\}$$
$$\mu_*(S') := \textstyle\bigcup_{\eta \in S'} \mu_*(\eta)$$

The set of thread states reachable in i transitions is denoted by $\tilde{S}^i := \mu_*(S^i) \subseteq \tilde{S}$. We propose to compute an over-approximation of $\tilde{S}^\infty$. This is sufficient to detect violations of reachability properties that are expressed in terms of the thread visible state[4], e.g., assertions.

Definition 5 ($\rightsquigarrow$): Let $\tilde{A} \subseteq \tilde{S}$ denote a set of thread states, and $\tilde{\zeta} \in \tilde{S}$ denote a thread state. Let $\tilde{A} \rightsquigarrow \tilde{\zeta}$ hold iff any of the following two conditions holds:

1) there is $\tilde{\eta} \in \tilde{A}$ such that there is a transition from $\tilde{\eta}$ to $\tilde{\zeta}$, i.e., $\tilde{\eta} \longrightarrow \tilde{\zeta}$,

[4]Note that the property still may depend on the behavior of multiple threads, due to the communication between the threads.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

$P(\tilde{\eta}.PC)$	PC	Ω
`skip`	$PC' = PC + 1$	$\Omega' = \Omega$
`goto` $\theta_1, \ldots, \theta_k$	$\bigvee_{i=1}^{k} PC' = \theta_i$	$\Omega' = \Omega$
`assume` e	$PC' = PC + 1$	$\Omega' = \Omega \ \wedge [\![e\,,\,\tilde{\eta}]\!] = \mathsf{true}$
$x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e	$PC' = PC + 1$	$\exists \iota.\ \Omega' = \Omega \quad [x_1/[\![e_1\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]$ $\ldots \quad [x_k/[\![e_k\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]\ \wedge [\![e\,,\,\tilde{\eta}\,,\,\tilde{\zeta}\,,\,\iota]\!]$
`start_thread` θ	$PC' = PC + 1$ $\vee PC' = \theta$	$\Omega' = \Omega$
`lock` l	$PC' = PC + 1$	$\Omega(l) = \mathsf{false} \wedge \Omega' = \Omega[l/\mathsf{true}]$
`unlock` l	$PC' = PC + 1$	$\Omega(l) = \mathsf{true} \wedge \Omega' = \Omega[l/\mathsf{false}]$

TABLE I

CONDITIONS ON THE EXPLICIT THREAD STATE TRANSITION $\tilde{\eta} \longrightarrow \tilde{\zeta}$ WITH $\tilde{\eta} = (PC, \Omega)$ AND $\tilde{\zeta} = (PC', \Omega')$, FOR VARIOUS STATEMENTS $P(PC)$, WHERE $\theta_i \in L$, e IS AN EXPRESSION AND $l \in \mathcal{L}$.

2) or there exists $\tilde{\eta} \in \tilde{A}$ and another transition out of $\tilde{A}$ with a disjoint set of locks that changes the global state of $\tilde{\eta}$ to that of $\tilde{\zeta}$. Formally, we require $\tilde{\zeta}.PC = \tilde{\eta}.PC$, $\tilde{\zeta} \overset{L}{=} \tilde{\eta}$ and there exist $\tilde{\eta}' \in \tilde{A}$ and $\tilde{\zeta}' \in \tilde{S}$ such that
   a) $\tilde{\eta}' \longrightarrow \tilde{\zeta}'$ with $\tilde{\eta}' \overset{G}{\neq} \tilde{\zeta}'$,
   b) $\tilde{\eta}' \overset{G}{=} \tilde{\eta}$, and
   c) $\tilde{\zeta}' \overset{G}{=} \tilde{\zeta}$,
   d) $l(\tilde{\eta}) \cap l(\tilde{\eta}') = \emptyset$ and $l(\tilde{\eta}) \cap l(\tilde{\zeta}') = \emptyset$.

   This case captures the communication between two threads.

We write $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ instead of the more cumbersome $\{\tilde{\eta}\} \rightsquigarrow \tilde{\zeta}$. Note that $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ implies $\tilde{A} \rightsquigarrow \tilde{\zeta}$ for any $\tilde{A}$ with $\tilde{\eta} \in \tilde{A}$.

Let $\tilde{T}^0 := \mu_*(S^0)$ denote the set of initial thread states, and $\tilde{T}^i$ for $i \in \mathbb{N}$ be defined recursively as follows:

$$\tilde{T}^i := \tilde{T}^{i-1} \cup \{\tilde{\zeta} \mid \tilde{T}^{i-1} \rightsquigarrow \tilde{\zeta}\}$$

The following claim holds by construction of $\rightsquigarrow$.

Theorem 1: For all $i \in \mathbb{N}_0$, the set $\tilde{T}^i$ is an over-approximation of the set of reachable thread states $\tilde{S}^i$.

Proof: The claim is shown by induction on i. For $i = 0$, the claim is trivial.

We show $\tilde{T}^i \supseteq \tilde{S}^i$ for the step from $i-1$ to i as follows. Let $\tilde{\zeta} \in \tilde{S}^i$. By definition of $\tilde{S}^i$, there is a full state $\zeta \in S^i$ and $u \in \{1, \ldots, \zeta.n\}$ such that $\tilde{\zeta} = \mu_u(\zeta)$. Furthermore, there exists $\eta \in S^{i-1}$ and $t \in \{1, \ldots, \eta.n\}$ such that $\eta \longrightarrow_t \zeta$. Let $\tilde{\eta}$ be a shorthand for $\mu_u(\eta)$. Using the induction hypothesis, we can conclude that $\mu_*(\eta) \subseteq \tilde{T}^{i-1}$, and in particular, $\tilde{\eta} \in \tilde{T}^{i-1}$.

If $u = t$, we have $\tilde{\eta} \longrightarrow \tilde{\zeta}$, which implies $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ (case 1 of Def. 5), and thus $\tilde{\zeta} \in \tilde{T}^i$, which concludes the claim.

If $u \neq t$, we make a case-split on the instruction $P(\eta.pc(t))$, which is executed in the transition from η to ζ (Table I):

- If $P(\eta.pc(t))$ is `skip`, `goto`, or `assume`, only the PC of thread t changes, and thus, $\tilde{\zeta} = \tilde{\eta}$, which implies $\tilde{\zeta} \in \tilde{T}^{i-1}$, and thus, $\tilde{\zeta} \in \tilde{T}^i$.
- If $P(\eta.pc(t))$ is `start_thread` and $u \neq \zeta.n$ (i.e., u is not the newly created thread), we also have $\tilde{\zeta} = \tilde{\eta}$, which concludes the claim. If $u = \zeta.n$, we have $\mu_t(\eta) \longrightarrow \tilde{\zeta}$, which concludes the claim.
- If $P(\eta.pc(t))$ is $x_1, \ldots, x_k := e_1, \ldots, e_k$ `constrain` e, let $k = 1$ without loss of generality. If $x_1 \in V_l$, only data local to thread t is modified, and the claim is shown as in case of `skip`. If $x_1 \in V_g$, let $v := \tilde{\zeta}.\Omega[x_1]$ denote the value that is assigned to x_1 by thread t. The new thread state $\tilde{\zeta}$ is equal to $\tilde{\eta}$ up to the assignment to x_1, i.e., $\tilde{\zeta}.PC = \tilde{\eta}.PC$ and $\tilde{\zeta}.\Omega = \tilde{\eta}.\Omega[x_1/v]$. Also, $\mu_t(\eta) \overset{G}{=} \tilde{\eta}$, and threads t and u hold a disjoint set of locks in state η. We therefore have $\tilde{\eta} \rightsquigarrow \tilde{\zeta}$ using case 2 of Def. 5.
- The statements `lock` and `unlock` are special cases of constrained assignments.

■

```
// Input: Boolean Program P with locations L,
// bad location b ∈ L
UNBOUNDEDTHREADAPPROXIMATION(P, b)
1  T̃:=μ*(S⁰);        // Initial States
2  while (true)
3        if (∃η̃ ∈ T̃.η̃.PC = b) return "Error state found";
4        F̃:={ζ̃ ∈ S̃ | T̃ ⇝ ζ̃};
5        if (F̃ ⊆ T̃) return "Property holds";
6        T̃:=T̃ ∪ F̃;
7  end
```

Fig. 1. High level description of the approximation algorithm for reachability in Boolean programs with unbounded threads

As the sequence $\tilde{T}^0, \tilde{T}^1, \ldots$ is monotonic and taken from a finite set, it has a fixed-point, and thus, $\tilde{T}^\infty$ is easily computable. The theorem above therefore gives rise to an algorithm (Fig. 1). If the algorithm terminates with "Property holds", the property is guaranteed to hold on the Boolean program. However, if an error state is found, there is no guarantee that the state is actually reachable. A counterexample trace can be computed by recording the one or two states that are used to compute a new thread state.

To illustrate the benefit of condition 2d) in Def. 5, consider the Boolean program in Fig. 2a, and assume a definition of $\rightsquigarrow$ without condition 2d). Suppose we start an unbounded number of threads that execute `f()`. The set of reachable thread states is shown in Fig. 2b. The lock protects the global variable, and

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

```
decl g, l;

void f() begin
 L1: lock l;
 L2: assert(!g);
 L3: g:=1;
 L4: g:=0;
 L5: unlock l;
 L6: skip;
end
```

(a)

PC	$\Omega(g)$	$\Omega(l)$
L1	0	0
L2	0	1
L3	0	1
L4	1	1
L5	0	1

(b)

Fig. 2. Boolean Program with critical section

thus, the assertion in `L2` holds.

We denote a state by (PC, g, l). However, because of $\{$(L5,0,1),(L2,0,1)$\} \rightsquigarrow$ (L2,0,0) $\longrightarrow$ (L3,0,0) and $\{$(L3,0,0),(L2,0,0)$\} \rightsquigarrow$ (L2,1,0), $\tilde{T}^{\infty}$ contains a state that violates the assertion, and we obtain a spurious error trace.

Remark 1: As an additional optimization, we keep track of whether a thread state was generated before or after a `start_thread` command. Thread states that are generated before the execution of any `start_thread` command need not participate in case 2 of definition 5. This optimization results in fewer spurious error traces, as the set of states of the program reachable up to the first `start_tread` command is no longer over-approximated. This case is omitted from the proof.

B. Refinement

The drawback of the over-approximation is that it may produce additional spurious counterexamples. Thus, for reachability properties φ, we may obtain $M' \not\models \varphi$ even though $M \models \varphi$ holds. If the algorithm generates an error trace, the error trace is simulated on the full model in order to rule out spurious error traces due to the imprecision introduced by $\rightsquigarrow$. Such a simulation corresponds to an incremental series of SAT instances, and is commonly performed by program analysis tools that implement abstraction refinement, e.g., SLAM and BLAST.

If the error trace is spurious, the over-approximation is refined. Note that we assume that this refinement is performed outside of the model checker as part of an abstraction refinement loop. Our algorithm may introduce spurious counterexamples due to the over-approximation caused by case 2 of Def. 5. The refinement algorithms in the existing predicate abstraction tools remove spurious traces by adding predicates to the model. This refinement strategy is effective for the over-approximation performed by our analysis algorithm, as the additional variables split states into two or more states $\tilde{\eta}, \tilde{\zeta}$ such that $\tilde{\eta} \overset{G}{\neq} \tilde{\zeta}$, which violates condition 2 of Def. 5.

IV. Symbolic Simulation

A. Symbolic State Representation

This section presents how thread states are represented symbolically. It extends the algorithm described in [24] to support an unbounded number of threads.

Definition 6: A symbolic formula is defined using the following syntax rules:

1) The Boolean constants true and false are formulae.
2) The non-deterministic choice identifiers $\star_1, \star_2, \ldots$ are formulae.
3) If f_1 and f_2 are formulae, then $f_1 \wedge f_2$, $f_1 \vee f_2$, and $\neg f_1$ are formulae.

The set of such formulae is denoted by $\mathcal{F}$.

A symbolic formula may evaluate to multiple values due to the choice identifiers. As an example, the pair of formulae $\langle \star_1, \star_2 \wedge \neg \star_1 \rangle$ may evaluate to $\langle 0,0 \rangle$, $\langle 1,0 \rangle$, $\langle 0,1 \rangle$, but not to $\langle 1,1 \rangle$. Given a particular valuation ι for the non-deterministic choices $\star_i$, we denote the value of a symbolic formula f as $[\![f]\!]^{\iota}$, i.e., $\iota \models f \iff [\![f]\!]^{\iota} = \mathsf{true}$.

We use these symbolic formulae in order to represent sets of explicit thread states:

Definition 7: A *symbolic thread state* $\tilde{\sigma}$ is a triple $\langle PC, \omega, \gamma \rangle$, with $PC \in L$, $\omega : V \mapsto \mathcal{F}$, and $\gamma \in \mathcal{F}$.

The first component of a symbolic thread state $\tilde{\sigma}$, namely PC, is identical to the first component of an explicit thread state (definition 2). The second component, called ω, is a mapping from the set of variables into the set of formulae. It denotes the *symbolic* valuation of the state variables. The last component, called γ, is a formula that represents the guard of the state symbolically, i.e., a constraint over the variables. Note that the program counter is represented explicitly, while the program variables are represented symbolically.

We can define the symbolic evaluation $[\![e\,,\,\tilde{\sigma}\,,\,\tilde{\tau}]\!]$ of an expression e in the symbolic thread state $\tilde{\sigma}$ and a next state $\tilde{\tau}$ in analogy to the definition for explicit states. The set of explicit thread states represented by a symbolic thread state $\tilde{\sigma}$ are those states $\tilde{\eta} \in \tilde{S}$ that satisfy the following conditions:

- They have the same PC: $\tilde{\eta}.n = \tilde{\sigma}.n \wedge \tilde{\eta}.PC = \tilde{\sigma}.PC$
- There exists a valuation ι that satisfies the guard γ and assigns values to the variables that match the values given by $\tilde{\eta}.\Omega$.

$$\exists \iota.\iota \models \gamma \ \wedge \ \forall v \in V.\ [\![v\,,\,\tilde{\eta}]\!] = [\![v\,,\,\tilde{\sigma}]\!]^{\iota} \qquad (1)$$

Note that the set of explicit states corresponding to a symbolic state is defined using a predicate in the parameter ι. Therefore, we have a *parametric representation* of the state-space. Parametric representations of sets of states have been used in formal verification before, e.g., in [26], [27], [28], but mostly in the context of hardware verification.

The construction of the symbolic thread states and the fixed-point loop with fixed-point detection using QBF follows the principle described in [24].

We also implement partial order reduction. In the context of algorithm 1, this corresponds to strengthening $\rightsquigarrow$ such that transitions are only propagated to thread states $\tilde{\zeta}$ that have a program counter $\tilde{\zeta}.PC$ that points to an instruction that either 1) reads one of the global variables begin modified or 2) writes a global variable.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

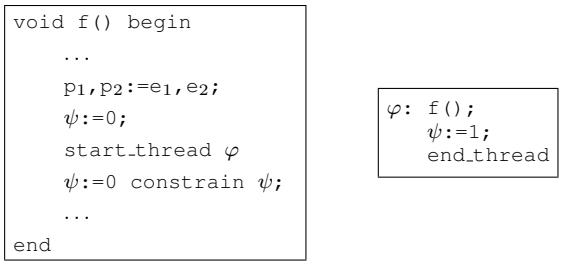


Fig. 3. Over-approximating a recursive call $f(e_1, e_2)$ with thread creation

B. Recursive Functions

Reachability for programs with recursion and concurrency (even with only two threads) is undecidable [7]. In order to model recursive programs we further extend the idea of conservative over-approximation.

Let f denote the function that is called, and let $p_1, \ldots, p_k \in V_l$ denote the parameters of the function. The expression e_i is passed as argument of the call for p_i.

- As first step, an assignment $p_1, \ldots, p_k := e_1, \ldots, e_k$ is performed.
- For synchronization upon return of the function, we introduce a new global variable ψ. An assignment statement is inserted before the function call that sets ψ to zero.
- The function call is replaced by a `start_thread` θ command, where θ denotes the first program location of f.
- After the function call, the statement $\psi := 0 \,\texttt{constrain}\, \psi$ is inserted. It sets ψ to false, but waits for ψ to become true before doing so.
- When f returns (using `return`), it sets ψ to true. The return values are passed by means of global variables.

The approximation of a recursive call $f(e_1, e_2)$ using thread creation is illustrated in Fig. 3.

This reduction is similar to an encoding of recursion commonly done in SPIN that uses a new channel for synchronization. In contrast to this reduction used for SPIN, we use a finite set of global variables for synchronization (one per call site), and therefore loose precision. The termination of the second recursion may synchronize with the call site of the first recursion and so on.

V. Experimental Results

We have implemented the technique described in this paper in a tool called BOPPO. To the best of our knowledge, there is no other model checker available for either Boolean programs with unbounded thread creation or concurrent Boolean programs with recursion. An implementation based on symbolic simulation has been compared to MOPED, SPIN, BEBOP, and ZING in [24]. However, none of these model checkers supports the class of programs the algorithm described in this paper aims at, which prevents experimental comparison. We make our implementation available to other researchers for experimentation[5].

The BOPPO is integrated as model checker for abstract models into SATABS, which is an implementation of SAT-based predicate abstraction [4], [29]. In this configuration, SATABS can verify safety properties of programs with (possibly unbounded) `while` loops that contain thread creating statements, e.g., the `pthread_create()` command. SATABS is also available for download[6].

The experiments have been performed on an Intel Xeon Processor with 2.8 GHz running Linux. The results are summarized in Table II. We use MiniSAT as our SAT-solver, and Quantor as QBF solver for the fixed-point detection. The run-time results are reported for our tool with symbolic partial order reduction and without symbolic partial order reduction. We also report the number of *symbolic* thread states that are explored, i.e., $|\tilde{T}|$. Note that one symbolic thread state typically corresponds to many explicit thread states, in particular if non-determinism is used heavily. On all experiments with non-trivial run-time, the run-time is dominated by the QBF solver.

We focus on the evaluation of the scalability of the implementation. We have two classes of benchmarks: artificial ones to measure scalability (ART series), and benchmarks extracted from the Apache httpd web-server package (AP series). The ART-PC-n series benchmarks are scaled in the number of program locations. Each benchmark generates an unbounded number of threads (using an infinite loop containing `start_thread`). Each thread then executes n non-deterministic assignments to global variables. The QBF instances generated for the fixed-point detection contain a number of quantified variables that is linear in n. The ART-V-n series benchmarks are parameterized in the number of variables, where n denotes the number of global (and thus, also thread-visible) variables. The number of variables in the QBF instances grows quadratically in n.

The APn series of benchmarks are extracted from an ANSI-C program using SATABS. While the original program generates a finite number of threads using the POSIX `pthread_create` command, the abstraction of the program generates an unbounded number of threads. The POSIX `pthread_mutex_lock` and `unlock` functions are mapped to `lock` and `unlock` in the Boolean programs. The various benchmarks correspond to different properties of the same program.

Apache (like most other programs) does not use locking during initialization, i.e., before it starts the worker threads. The algorithm as described above results in states with inconsistent global predicates, which produces a large number of spurious counterexamples. For this benchmark, we therefore extend the algorithm to distinguish two different types of thread states using a flag as suggested in remark 1 above. The flag is false in the initial state, and is set to true upon execution of `start_thread`. Case 2 of Def. 5 is changed such that global data is only passed between thread states that have the same value of the flag. After the initialization phase, most writes to global data are protected by means of locks, which

[5] `http://www.verify.ethz.ch/boppo/`

[6] `http://www.verify.ethz.ch/satabs/`

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

Benchmark	Without PO		With PO	
	Time	$\#\tilde{\sigma}$	Time	$\#\tilde{\sigma}$
ART-PC-10	6.0s	893	<0.1s	21
ART-PC-20	21.0s	2723	<0.1s	31
ART-PC-100	⋆		0.2s	111
ART-PC-1000	⋆		8.3s	1011
ART-V-10	17.2s	801	0.1s	29
ART-V-20	111.7s	2571	0.3s	49
ART-V-100	⋆		5.3s	209
ART-V-1000	⋆		3508.1s	2009
AP1	⋆		242.7s	8009
AP2	⋆		269.5s	10766
AP3	⋆		288.9s	11422
AP4	⋆		155.1s	5453
AP5	⋆		1130.9s	43812

TABLE II

SUMMARY OF RESULTS: A STAR DENOTES THAT THE TWO HOUR TIMEOUT WAS EXCEEDED. THE COLUMNS UNDER $\#\tilde{\sigma}$ CONTAIN THE NUMBER OF SYMBOLIC THREAD STATES.

prevents spurious error traces.

On the artificial examples, the regular refinement with WPs works fine to eliminate the spurious CEs, as it adds more Boolean variables, which make the states different (and only states with equal values of the global variables participate in case 2 of Def. 5).

VI. CONCLUSION

CEGAR-based symbolic model checkers have proven themselves tremendously useful for sequential programs. For shared-memory concurrent software they have been effectively useless. This is due to fact that the underlying tool that checks the abstractions must always return an answer—something that no tool has been able to guaranteed when applied to abstractions that can support arbitrary thread creation. This paper introduces a new symbolic model checker for software abstractions (Boolean programs) that supports arbitrary thread creation while guaranteeing termination. This checker can potentially return spurious counterexamples, but it *is always able* to produce one.

REFERENCES

[1] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer, "Thread modular abstraction refinement," in *CAV*, ser. LNCS. Springer, 2003, pp. 262–274.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," in *EuroSys*, 2006.

[3] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular verification of software components in C," in *International Conf. on Software Engineering*, 2003, pp. 385–395.

[4] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate abstraction of ANSI–C programs using SAT," *Formal Methods in System Design*, vol. 25, pp. 105–127, September–November 2004.

[5] M. K. Ganai, A. Gupta, and P. Ashar, "DiVer: SAT-based model checking platform for verifying large scale systems," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 575–580.

[6] M. Rinard, "Analysis of multithreaded programs," in *SAS*, ser. LNCS, vol. 2126. Springer, 2001. [Online]. Available: citeseer.ist.psu.edu/article/rinard01analysis.html

[7] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 2, pp. 416–430, March 2000.

[8] O. Burkart and B. Steffen, "Composition, decomposition and model checking of pushdown processes," *Nordic Journal of Computing*, vol. 2, pp. 89 – 125, 1995.

[9] J. Esparza and S. Schwoon, "A BDD-based model checker for recursive programs," in *CAV*, ser. LNCS 2102. Springer, 2001, pp. 324–336.

[10] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for Boolean programs," in *SPIN 00*, ser. LNCS 1885. Springer, 2000, pp. 113–130.

[11] S. Chaki, E. M. Clarke, N. Kidd, T. W. Reps, and T. Touili, "Verifying concurrent message-passing C programs with recursive calls." in *TACAS*, ser. LNCS, vol. 3920. Springer, 2006, pp. 334–349.

[12] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent software," in *TACAS 05*. Springer, 2005.

[13] E. Clarke, M. Talupur, T. Touili, and H. Veith, "Verification by network decomposition," in *CONCUR 04*, ser. LNCS, vol. 3170. Springer, 2004, pp. 276–291.

[14] C. Ip and D. Dill, "Verifying systems with replicated components in murϕ," in *CAV*, ser. LNCS, vol. 1102. Springer, 1996, pp. 147–158.

[15] A. Pnueli and T. Arons, "TLPVS: A PVS-based LTL verification system," in *Verification–Theory and Practice: Proceedings of an International Symposium in Honor of Zohar Manna's 64th Birthday*, ser. LNCS. Springer, 2003, pp. 84–98.

[16] T. A. Henzinger, R. Jhala, and R. Majumdar, "Race checking by context inference." in *PLDI*. ACM, 2004, pp. 1–13.

[17] C. Flanagan and S. Qadeer, "Thread-modular model checking." in *SPIN*, ser. LNCS, vol. 2648. Springer, 2003, pp. 213–224.

[18] J. E. Ahmed Bouajjani and T. Touili, "Reachability analysis of synchronized PA systems," in *Infinity 04: International Workshop on Verification of Infinite-State Systems*, 2004.

[19] S. Stoller, "Model-checking multi-threaded distributed Java programs," in *SPIN 00*. Springer, 2000.

[20] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2(4), 2000.

[21] E. Yahav, "Verifying safety properties of concurrent java programs using 3-valued logic." in *POPL*. ACM Press, 2001, pp. 27–40.

[22] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie, "Zing: Exploiting program structure for model checking concurrent software," in *CONCUR 04*, 2004.

[23] T. Ball, S. Chaki, and S. K. Rajamani, "Parameterized verification of multithreaded software libraries," in *TACAS*. Springer, 2001.

[24] B. Cook, D. Kroening, and N. Sharygina, "Symbolic model checking for asynchronous boolean programs," in *SPIN*. Springer, 2005, pp. 75–90.

[25] G. Holzmann and D. Peled, "The State of SPIN," in *CAV*, ser. LNCS. Springer, 1996, vol. 1102, pp. 385–389.

[26] P. Jain and G. Gopalakrishnan, *IEEE Transactions on Computer-Aided Design*, vol. 13, 1994.

[27] O. Coudert and J. Madre, "A unified framework for the formal verification of sequential circuits," in *ICCAD*. IEEE, 1990, pp. 78–82.

[28] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, "Formal verification using parametric representations of boolean constraints," in *DAC*. ACM Press, 1999, pp. 402–407.

[29] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, ser. Lecture Notes in Computer Science, vol. 3440. Springer Verlag, 2005, pp. 570–574.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

An Improved Distance Heuristic Function for Directed Software Model Checking

Neha Rungta
Department of Computer Science
Brigham Young University
Provo, UT 84601
Email: neha@cs.byu.edu

Eric G Mercer
Department of Computer Science
Brigham Young University
Provo, UT 84601
Email: egm@cs.byu.edu

Abstract— **State exploration in directed software model checking is guided using a heuristic function to move states near errors to the front of the search queue. Distance heuristic functions rank states based on the number of transitions needed to move the current program state into an error location. Lack of calling context information causes the heuristic function to underestimate the true distance to the error; however, inlining functions at call sites in the control flow graph to capture calling context leads to an exponential growth in the computation. This paper presents a new algorithm that implicitly inlines functions at call sites to compute distance data with unbounded calling context that is polynomial in the number of nodes in the control flow graph. The new algorithm propagates distance data through call sites during a depth-first traversal of the program. We show in a series of benchmark examples that the new heuristic function with unbounded distance data is more efficient than the same heuristic function that inlines functions at their call sites up to a certain depth.**

I. Introduction

Multi-core processor design and hyper-threading create a need for techniques to validate concurrent interactions in threaded software artifacts. Traditional validation techniques based on test vector generation generally break down in the presence of concurrency since they cannot control scheduling decisions imposed by the operating system when running the input vectors. As a consequence, the validation is not effective in discovering subtle race or deadlock conditions that often lead to unexpected program behavior. Model checking is particularly effective in finding errors in deep execution traces because it considers all possible thread schedules in its analysis. Model checking has the potential to aid software validation if it can be effectively applied to real software artifacts.

State explosion is inherent in model checking, and it is especially problematic in software model checking because of the size and complexity of typical software artifacts. The process of model checking systematically explores the behavior space of the program in some way. There are several different tools and approaches to address the state explosion problem in software model checking, [1]–[5], and the work in this paper specifically focuses on directed model checking [6]–[8].

Directed model checking guides the search into areas of the state space where errors are more likely to exist. It aims to find a property violation before computation resources are exhausted. Directed model checking uses a heuristic function to rank successor states during state space exploration. The search order follows the ranking of states on the search frontier using a priority queue rather than a search stack. An *accurate heuristic function* reduces the number of states generated before error discovery without dramatically decreasing the frequency of state generation.

Early heuristics use notions from circuit design technology for computing the distance estimate. For example, hamming distance heuristics use the explicit state representation to estimate a bit-wise distance between the current state and an error state [9]. Current heuristic functions for directed software model checking are broadly classified into two categories: property based heuristics and structural heuristics. A property based heuristic function tries to estimate the number of changes in the program values needed to violate a property, while structural heuristics consider the structure of either the actual program or its resulting transition system to compute the heuristic estimate. Examples of property based heuristics are in [10], [11]. The work in this paper focuses on structural heuristics.

The notion of structural heuristics is introduced in [12]. The heuristics in [12] exploit the structural properties of thread interdependencies specific to only Java programs to find concurrency errors. Distance heuristic functions [13]–[15] are structural heuristics that compute the minimal number of transitions required to reach an error location from the current state in the control flow representation of the artifact. These heuristic functions have been shown to be effective in driving threads into race and deadlock conditions.

The extended-FSM (EFSM) distance heuristic combines statically computed distance estimates from the structure of the software artifact with the dynamic call trace in the run-time stack extracted from the state representation of the software artifact to improve the accuracy of the heuristic values [15]. The heuristic function is based on the following notion: at a given program location, the program is either going to reach a return point for the callee without encountering an error and return to the caller; or it encounters an error before it reaches the return point and does not return to the caller. The algorithm to compute the EFSM distance heuristic uses a graph with bounded calling context to compute the distances

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

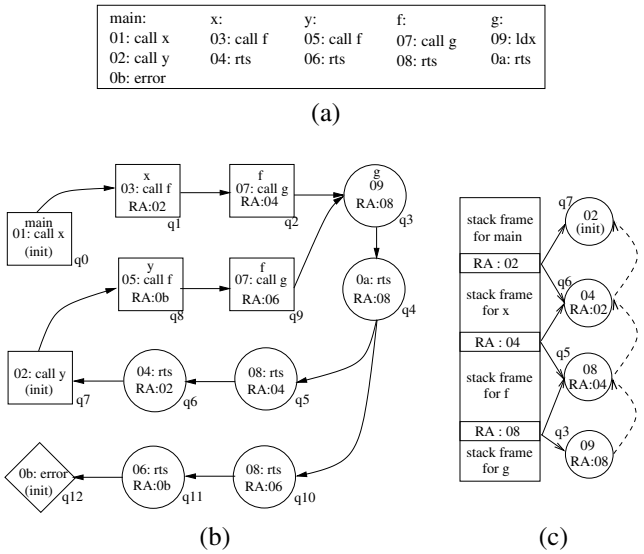


Fig. 1. A program and analysis demonstrating underestimation in the EFSM heuristic due to bounded calling context. (a) A program with a nested call depth of four. (b) An one-bounded CFG for the program that inlines procedures at call sites. (c) The run-time stack for a given state of the program.

in the forward direction. To build such a graph, procedures are inlined at call sites up to the depth of the user specified bound. Although the EFSM heuristic function reduces the number of states before error discovery in various examples, it is not an efficient heuristic function because the time required to construct the bounded graph increases exponentially with the bound; thus, the heuristic function does not scale well to programs with deeply nested function calls where large bounds are required for accurate heuristic estimates.

This paper presents a new *full context aware* (FCA) algorithm that implicitly inlines functions at call sites to compute distance data with unbounded calling context that is polynomial in the number of nodes in the *control flow graph* (CFG) for the software artifact. The new algorithm computes full context information for non-recursive programs with resolved function pointers, and works by propagating distance data through call sites during a depth-first traversal of the program's CFG. We show, in a series of benchmark examples, that a new heuristic function, e-FCA that is based on the EFSM heuristic but uses the FCA for forward distance estimates, generates fewer states and decreases total running time compared to the EFSM heuristic function that uses the inlined bounded calling context information [15].

II. Motivating Example

We demonstrate with an example how the accuracy of the EFSM heuristic in [15] relies on the bounded calling context used while computing the static shortest-path distances for the forward analysis. A program with a maximum possible call depth of four is presented in Fig. 1(a) where an error location, $0b$, is reachable from the `main` procedure only after making a call to procedures `x` and `y`. Procedures `x` and `y` both make calls to procedure `f` which in turn calls procedure `g`. In

```
procedure Extended_FSM(pc, rstack)
 1: d = 0, D_e = ∅
 2: while (rstack) do
 3:     ret_locs = get_entries(rstack, k)
 4:     n = get_node_in_k_bounded_CFG(pc, ret_locs)
 5:     E = {FSM(n, n_e) + d | n_e ∈ Errors ∧ in_scope(n_e, n)}
 6:     D_e = D_e ∪ E
 7:     n_end := return_statement(n)
 8:     d = d + FSM(n, n_end) + 1
 9:     pc = rstack.top()
10:     rstack.pop()
11: return min(D_e)
```

Fig. 2. Pseudo-code for the EFSM algorithm.

our program, the EFSM heuristic function tries to accurately estimate the minimal number of transitions required to reach the error location from the current program location. For example, from line 01 of `main`, it computes the number of instructions that need to be executed in order for the program state to reach the error location in line $0b$ of `main`. Ideally, the heuristic computation needs to account for the fact that the true execution flow of the program moves through procedures `x` and `y` before reaching the error.

The EFSM heuristic inlines procedures at call sites up to a bounded depth to capture partial context information. It does this by constructing a k-bounded CFG in a depth-first traversal of the program, where k is the specified bound. Each node in a k-bounded CFG is a location in the program with up to k entries for the partial call trace of length k used to arrive at that location. A one-bounded CFG for the program in Fig. 1(a) is shown in Fig. 1(b), where boxes represent call sites, circle nodes are return points or arbitrary program instructions, and the diamond shape nodes represent the error locations in the artifact. Each node, regardless of its type, has a program location identifier to map it back to the original program followed by a return address indicated by the `RA` label. There is a single return address in each node for this example because the k-bound of the graph is one. Returning to our example, procedures `x`, `y`, and `f` have enough context information to be uniquely inlined at their call sites; however, procedure `g` is not fully inlined at its call sites because unlike procedure f that is called from two unique call sites: locations 03 and 05, while procedure `g` is invoked two times from the same call site: location 07. In Fig. 1(b), the node q_4, an `rts` (return) instruction, is at program location $0a$ in procedure `g`. The return instruction can transfer control to any node that is at location 08 in procedure `f`. The edges from the q_4 node show that there are two possible return points: nodes q_5 and q_{10}. Both are at location 08, and both are an invocation of `f`. Without a k-bound of at least two, there is not enough context to create a unique invocation in the graph of the partial call trace, `f` $\rightarrow$ `g`, for both the `x` and `y` originations. The missing context leads to an underestimation of the final estimate in a shortest-path analysis because the shortest-path analysis uses the `x` invocation to get to procedure `f` but returns to the `y` invocation.

The EFSM heuristic function in [15] uses the calling context in the run-time stack present in the state of the program to recapture part of the missing calling context in the k-bounded CFG. There is no additional overhead in maintaining the run-time stack of the program because at any point in the program, the model checker has a complete snapshot of the actual state of the program including its entire run-time stack. The concrete run-time stack reflects the complete call trace from the top-most procedure of the artifact to the current program location which can be used in conjunction with the k-bounded graph to produce an accurate distance estimate. This is done by unrolling the run-time stack, and at each stack frame, considering the case that the program moves forward to encounter an error without returning from the current stack frame, or it returns from the current stack frame and then moves forward to encounter an error. The heuristic function minimizes over each of these scenarios as it moves through stack frames.

In Fig. 1(c), we present an example of a run-time stack from the concrete state of the program in the model checker. The current stack frame is for procedure `g` shown at the bottom of the run-time stack since it grows downward. The return address for the current procedure in the run-time stack is location 08 in procedure `f`. The current program location is 09 in the procedure `g`. The EFSM heuristic function, shown in Fig. 2, takes the current program location (pc) and combines it with the first k return locations (ret_locs) from the run-time stack ($rstack$) to identify the node (n) corresponding to the current program state in the k-bounded CFG (lines 3-4). The corresponding node in the one-bounded graph for the current program state of Fig. 1(b) is q_3 since it is at program location 09 with a return address of 08. The heuristic function in Fig. 2 now computes a distance estimate in the forward direction within the scope of the current stack frame (line 6). It requests a distance estimate on the k-bounded graph to all possible errors ($n_e \in Errors$) in the forward direction using a shortest-path analysis ($\mathsf{FSM}(n, n_e)$) assuming the current procedure does not return from the current stack frame ($\mathsf{in\_scope}(n_e, n) = true$). In our example, the error is not reachable from procedure `g` without moving through its return point so the shortest-path analysis returns ∞. The heuristic function in Fig. 2 then makes a note of this distance (line 6) and then computes the shortest distance to the previous call frame through the return statement of the current procedure (lines 7-8). It finally simulates returning from the current call frame by making the first return location its current program location (line 9).

Continuing with the concrete example in Fig. 1(c), after returning from procedure `g`, the EFSM heuristic function combines the new current program location, now 08, and the return location, 04, to find the next node in the k-bounded CFG. This corresponds to node q_5 in the one-bounded graph in Fig. 1(b). The heuristic function requests another forward estimate which is still ∞. It then considers the cost of returning from the stack frame. The algorithm in Fig. 2 repeats this process until it runs out of stack frames in the run-time stack (line 2), and it then reports the distance estimate to the nearest error computed during the analysis (line 11). The key observation in this example is that the call site for procedure `g` is resolved using the run-time stack, and the EFSM heuristic reports the correct distance value from node q_3 to the error location. This does not, however, completely remove the underestimation in other distance estimates.

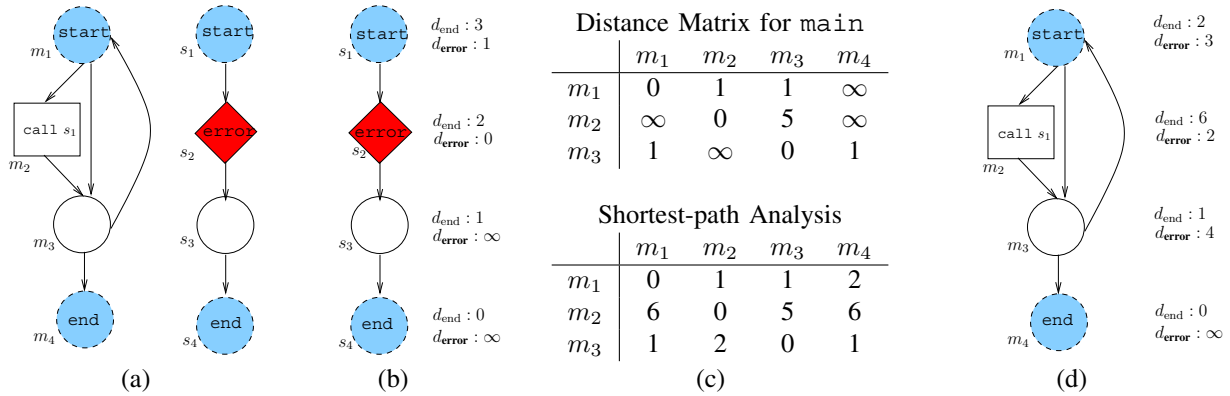
When the program execution is at the topmost level of the call structure or has a shallow call depth, the heuristic estimate computation is reduced to a shortest-path analysis on the k-bounded CFG. Returning to our example, suppose the concrete state in the model checker has a single frame in the run-time stack showing the current location to be node q_0. A request for the distance estimate in the forward direction returns a distance of seven which is the shortest-path to the error node q_{12} from node q_0 in the one-bounded CFG. The missing context information at depths greater than one is needed to resolve the unique call sites leading to node q_4, and it causes the shortest-path analysis to choose a path that is not consistent with the actual program execution from node q_0.

The cost of building the k-bounded graph is exponential in the nested call depth due to inlining. The cost of doing the shortest-path analysis is also very expensive since it is run several times to account for the scoping check on the path to the error. For the EFSM heuristic to be accurate, it needs full calling context, but for it to be efficient in runtime, it needs a small k-bound. The goal of this work is to produce an efficient estimate in terms of its accuracy and computational overhead.

III. Full Context Aware (FCA) Algorithm

The FCA algorithm is an interprocedural control flow analysis technique that implicitly constructs call traces to compute static lower-bounds on distance estimates to return locations of the procedures and nearest error locations in a software artifact using shortest-path analysis and depth-first traversal. The FCA algorithm uses the reverse invocation order to summarize the shortest-path analysis in all the callees of a given procedure. It then propagates the summarized distance information of the callees back to the given procedure. The input to the algorithm is a set of CFGs with a single CFG for each method or procedure in the artifact. Fig. 3(a) is an example of the input for a program with two procedures `main` and `sub1`. Each CFG has a single start node and end node. A call node is represented as a box in the CFG. The label in the call node identifies the start node for the target CFG of the call. The diamond shape nodes represent error nodes in the program. These are most often critical sections or assertion points in the software artifact. We associate values with each CFG node for the distance to the end node (d_{end}) and the distance to the nearest error node (d_{error}).

The FCA algorithm uses a depth-first traversal to build a distance matrix for a given CFG to use in a shortest-path analysis to compute d_{end} and d_{error} for each node. We associate with each individual CFG a distance matrix, L, that is defined over the number of nodes in the CFG. L is initialized with entries along the diagonal set to zero and all other entries



Distance Matrix for `main`	m_1	m_2	m_3	m_4
m_1	0	1	1	∞
m_2	∞	0	5	∞
m_3	1	∞	0	1

Shortest-path Analysis	m_1	m_2	m_3	m_4
m_1	0	1	1	2
m_2	6	0	5	6
m_3	1	2	0	1

Fig. 3. Execution of the FCA on a set of CFGs. (a) A set of two CFGs for a software artifact. (b) The CFG for `sub1` annotated with d_{end} and d_{error} data. (c) Distance matrix L for `main` before and after shortest-path analysis. (d) The CFG for `main` annotated with d_{end} and d_{error} data.

set to ∞. Edge costs in L are added as the depth-first traversal moves through nodes in the CFG. The depth-first traversal begins at the start node of the top-level method in the software artifact. In our example, the traversal starts at node m_1, updates the (m_1, m_2) entry in L to add the cost of the $m_1 \rightarrow m_2$ edge, and then visits m_2. The distance between two immediate successors in a CFG is one for all nodes except the successors of call nodes. The distance to the immediate successor of a call node needs to reflect the cost of moving through the target CFG of the call.

The depth-first traversal moves to the start node of the target CFG at a call node, and when the traversal returns, it then explores the immediate successor of the call node in the current CFG. The distance to the immediate successor of the call node relies on the analysis of the target CFG to account for the cost of moving through the target CFG without encountering an error. The distance to the immediate successor of the call node is the d_{end} value stored in the start node of its target CFG plus two: one to move to the start node of the target CFG and one to return from the end node of the target CFG. Fig. 3(b) shows the d_{end} and d_{error} values in the `sub1` CFG when the traversal returns to m_2 after analyzing `sub1`. Recall that call node m_2 points to the s_1 start node. The d_{end} value for s_1 is three. This reflects the number of program steps required in `sub1` to reach its s_4 end node. The traversal updates the (m_2, m_3) entry in L by setting the $m_2 \rightarrow m_3$ edge to five (three plus two), and continues the traversal by visiting node m_3. The top matrix in Fig. 3(c) is the final L matrix for `main`. The matrix includes all the edges in the `main` CFG with the cost of moving through `sub1` included in the row for m_2. Note that the matrix excludes the row for m_4 since the end node has no successors in the traversal. The L matrix provides the requisite data to derive d_{end} and d_{error} values for the CFG nodes.

The FCA algorithm computes d_{end} for each node in the current CFG with a shortest-path analysis using the distance matrix when the traversal is ready to backtrack out of the start node. Recall that L only includes nodes in the immediate CFG since call nodes move to the immediate successor in the CFG with the cost of moving through the target CFG. The bottom matrix in Fig. 3(c) is the final L matrix for `main` after the shortest-path analysis. Each d_{end} value for the CFG of `main` is set to its corresponding entry in the m_4 column of L after the shortest-path analysis. This is the shortest-path through the CFG to the end node m_4 including the cost of function calls.

The nearest error distance values are computed by minimizing over distances to error locations in the current CFG and distances to error locations reachable from the target CFGs of call nodes. In the discussion, N_{error} and N_{call} denote the sets of error locations and call nodes respectively for a given CFG. We use the results of the shortest-path analysis in the distance matrix, L, to compute distances to the nearest error location. For convenience, a global array, $D_{\text{error}}(n)$ is used to store the distance to the nearest error, d_{error}, for node n in a CFG. The equations for computing d_{error} for a given node n are

$$d_{\text{local}} = \min_{n_e \in N_{\text{error}}} (L(n, n_e)) \quad (1)$$

$$d_{\text{nonlocal}} = \min_{n_c \in N_{\text{call}}} (L(n, n_c) + D_{\text{error}}(n'_{\text{start}}) + 1) \quad (2)$$

$$d_{\text{error}} = \min(d_{\text{local}}, d_{\text{nonlocal}}) \quad (3)$$

$$D_{\text{error}}(n) = d_{\text{error}} \quad (4)$$

where n is the current node being analyzed and a call node n_c points to the start node n'_{start} of its target CFG.

The d_{local} value in Equation 1 is the distance to the nearest error in the immediate CFG. This error is reachable without having to move into a different CFG through a call node. The value is taken directly from the shortest-path analysis results in the distance matrix ($L(n, n_e)$ is the shortest-path from node n to node n_e in the current CFG).

The d_{nonlocal} value in Equation 2 is the distance to the nearest error through a call node of the immediate CFG. This error is only reachable by moving into a different CFG through a call node. The equation computes the transitive distance of first moving forward to the call node and then moving from the start node of the target CFG to the error. In the equation for the n_c call node, n'_{start} is the start node of its target CFG. The value stored in $D_{\text{error}}(n'_{\text{start}})$ is computed prior by virtue of the depth-first traversal. The traversal only triggers the distance

```
procedure compute_distances(N, E, n_start, n_end, N_error, N_call)
 1: /* Visited is global variable initialized to ∅ */
 2: if n_start ∉ Visited then
 3:     Visited = Visited ∪ {n_start}
 4:     /* L : N × N → ℕ ∪ {∞}, entries along the diagonal are set
        to 0, while all other entries are set to ∞ */
 5:     L = traverse_CFG(n_start, N_call, L)
 6:     L = compute_all_pairs_shortest_distance(L)
 7:     for each n ∈ N do
 8:         d_end = L(n, n_end)
 9:         /* D_end, a global array of size X is initialized to 0 */
10:         D_end(n) = d_end
11:         d_local = min_{n_e ∈ N_error}(L(n, n_e))
12:         ⟨N', E', n'_start, n'_end, N'_error, N'_call⟩ = Target(n_k)
13:         d_nonlocal = min_{n_k ∈ N_call}(L(n, n_k) + d_error(n'_start) + 1)
14:         d_error = min(d_local, d_nonlocal)
15:         /* D_error, a global array of size X is initialized to 0 */
16:         D_error(n) = d_error
17: return
18:
procedure traverse_CFG(n_x, N_call, L)
19: if n_x ∈ N_call then
20:     ⟨N', E', n'_start, n'_end, N'_error, N'_call⟩ = Target(n_x)
21:     compute_distances(N', E', n'_start, n'_end, N'_error, N'_call)
22:     d_succ = D_end(n'_start) + 2
23: else
24:     d_succ = 1
25: /* Conditional branches have multiple successors */
26: for each n'_x ∈ succ(n_x) do
27:     L(n_x, n'_x) = d_succ
28:     L = traverse_CFG(n'_x, N_call, L)
29: return L
```

Fig. 4. Pseudo-code for the FCA algorithm.

analysis when it is ready to backtrack out of a start node to resolve call dependencies in the computation. The final d_{error} value for node n in Equation 3 is either local to the CFG or reached through a call node in the CFG.

Fig. 3(d) shows the end results of the FCA algorithm for `main`. Fig. 3(b) and Fig. 3(d) give the complete view of the final analysis. For Fig. 3(d), the path to the end node for m_1 bypasses the m_2 call node for a distance of two, which is the number of edges in the path $m_1 \rightarrow m_3 \rightarrow m_4$. An error location is only reachable through the target CFG of the call node m_2. The nearest error for m_1 is three which represents the path $m_1 \rightarrow m_2 \rightarrow s_1 \rightarrow s_2$ in the CFGs. An error location cannot be reached from node m_4.

The FCA algorithm lower-bounds all distance estimates by assuming shortest-paths through CFGs. From this, the d_{error} and d_{end} data by themselves form an admissible and consistent distance estimate similar to the finite state machine (FSM) distance heuristic in [13]. Regardless of the true path of execution, the length of that path is at least that of the shortest-path through the CFG. An example is seen in Fig. 3(d) where the algorithm bypasses the m_2 call node to reach the m_4 end node. If the true path of execution follows m_2, then the actual distance is strictly larger than the reported distance. This lower-bound also appears in all iterative constructs of the CFG.

The pseudo-code for the FCA algorithm is presented in Fig. 4. In Fig. 4, a *CFG* is a tuple $\langle N, E, n_{\text{start}}, n_{\text{end}}, N_{\text{call}}, N_{\text{error}} \rangle$ where N is set of uniquely labeled nodes, $E \subseteq N \times N$ is the set of edges, $n_{\text{start}} \in N$ is a unique start node, $n_{\text{end}} \in N$ is a unique end node, $N_{\text{call}} \subseteq N$ is a set of call nodes, and $N_{\text{error}} \subseteq N$ is a set of error nodes. D_{end} and D_{error} are global arrays that store distances to the end node and nearest error node respectively for $X = |\bigcup_{1 \leq i \leq m} N_i|$ nodes where m is the number of procedures in the artifact. Note that the d_{error} and d_{end} values stored in D_{error} and D_{end} arrays are the same d_{error} and d_{end} values annotated on the CFGs. The function Target takes a call node as input and returns the target CFG of the call node. Finally, the function $\text{succ}(n_x) = \{n_y \in N \mid (n_x, n_y) \in E\}$, which means the succ function returns a set containing all the immediate successors of the input node, n_x.

In Fig. 4, the compute_distances function intializes a distance matrix L (line 4) for the input CFG and calls the traverse_CFG function (line 5). The traverse_CFG function uses a depth-first traversal to add edge costs between successors of the CFG in the distance matrix L (lines 26-28). If the traversal encounters a call node (line 19), the algorithm makes a mutually recursive call to compute_distances with the target CFG of the call node (line 21). When the execution returns, it adds the edge cost to the immediate successor of the call node taking into account the cost of moving through the target CFG without encountering an error (line 22). For all other nodes, the distance between two immediate successors in a CFG is one (line 24). After the traversal of the CFG is done, the algorithm returns the distance matrix L (line 29), and the flow of execution returns to the compute_distances function where the FCA algorithm computes the all-pairs shortest-path on the distance matrix L (line 6). For each node in the CFG, it adds the corresponding d_{end} and d_{error} values to the global arrays D_{end} and D_{error} (lines 7-16). The d_{error} value is computed by minimizing over distances to error locations in the current CFG (d_{local}) and distances to error locations reachable from the target CFGs of call nodes (d_{nonlocal}).

The intial traversal and the final algorithm to propagate the d_{end} and d_{error} values are linear in the number of nodes in the artifact since the traversal reuses the information from the secondary analysis if it encounters the same CFG numerous times from different call nodes in the artifact. The complexity of the secondary analysis is $O(X^3)$, where X is the total number of nodes in the artifact because an all-pairs shortest-path algorithm is run once for every reachable CFG in the artifact. Hence, the complexity of the FCA algorithm is polynomial in time and space with regards to the total number of nodes in the CFGs as a result of the shortest-path analysis.

IV. EFSM WITH FCA

The FCA is a forward analysis algorithm which is run once statically. The FCA cannot resolve the non-determinism arising from the end nodes in the CFGs of the program. For example, in Fig. 1(a) there are two calls to procedure `f`. The CFG of `f` does not contain any information about where the flow of execution returns when it exits procedure `f`. The only information present in the nodes of the CFGs are the distance values to the end (d_{end}) of the CFGs and the distance values to error locations (d_{error}) in the forward direction without

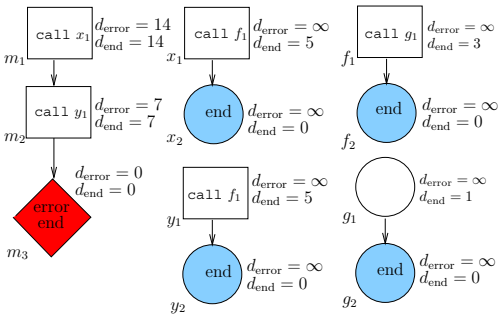


Fig. 5. The CFGs for the program in $Fig.\ 1(a)$ annotated with context aware distances after running the FCA algorithm.

executing the end of the CFGs. We dynamically recreate the call trace based on the values of the run-time stack like the EFSM heuristic to compute true successors of end nodes for a particular program execution path.

For the program shown in Fig. 1(a), the CFGs for each procedure annotated with d_{error} and d_{end} values after executing the FCA computation are shown in Fig. 5. Now let us consider a concrete example of how the EFSM is combined with the FCA to compute accurate heuristic estimates. Suppose the values on the return stack are: $\langle x_2, m_2 \rangle$ where the x_2 is the first return location encountered on exiting the current stack frame, and m_2 is the next return location. The current location of the program is f_1 in procedure `f`. The heuristic function in the EFSM requests a distance estimate to the error in the forward direction without exiting from the current procedure. Instead of computing this estimate on a k-bounded graph, it now considers the d_{error} value present on node f_1. In Fig. 5, we can see that the value of d_{error} is ∞ for node f_1 which means the error is not reachable in the forward direction.

After noting the value of d_{error} at node f_1, the heuristic function uses the value of d_{end} to compute the distance to the end node of procedure `f`. It requires this value to estimate the distance to the previous call frame. The algorithm to compute the heuristic estimate has an accumulator variable, $path$, that keeps track of the cost incurred in backtracking through the call frames. In Fig. 5 the value of d_{end} is three; the heuristic function adds one to it to account for the return and sets $path$ equal to four. Next, the heuristic simulates returning from the current call frame by making the first value on the return stack, x_2, its current program location like the EFSM algorithm. The value of d_{error} on node x_2 is ∞ showing that the error is still not reachable. Since x_2 is an end node, the value of d_{end} is zero, and the heuristic function increments $path$ by one for the return, changing the value of $path$ to five. The heuristic function repeats the process of unrolling the stack by moving to node m_2, where the value of d_{error} is seven. It adds this value to $path$ to get the final value of twelve as the estimate of the distance to the error. At all points of forward computation to find the error location, and while going to the end node, the heuristic function has access to unbounded context information from the FCA algorithm resulting in a better lower-bound on the true estimate of the distance to the error which is also admissible and consistent.

TABLE I
TIME TAKEN IN SECONDS FOR STATIC ANALYSIS

Name	T	M	k	FSM	EFSM	FCA
Hyman	2	3	1	0	3	0
Hyman	2	4	1	1	11	0
Hyman	2	5	1	1	27	0
D-phil	2	2	1	1	76	0
D-phil C	2	2	1	0	76	0
D-phil	2	3	1	1	146	0
D-phil C	2	3	1	0	147	0
D-phil	2	4	0	1	4	1
D-phil C	2	4	0	1	4	1
D-phil	2	5	0	2	7	1
D-phil C	2	5	0	2	7	1
Barbers	3	3	1	1	41	0
Barbers	3	4	0	1	3	1
Barbers	3	5	0	1	4	1

To calculate the heuristic estimate for a concurrent program with multiple threads, the approach presented in [13] computes the distance to the nearest error location for each thread and sums up the individual estimates to create a final heuristic value. To ensure underestimation of the distance to the error, we modify the number of individual estimates summed together based on the property being verified. For example, the heuristic estimate for a $mutex$ violation is the sum of distances in two threads which have the shortest paths to the critical section compared to all the other threads. Now, consider the property which is a check to see if any thread reaches a certain location in the program, like an $assert$ statement. In such a case, the heuristic estimate is the smallest distance in the set of estimated distances from the current location to the desired location for each thread. Another useful property checked in concurrent programs is whether two or more threads are $deadlocked$. In this case, we take the summation of the distances from the current state to the error state for two or more threads that can lead to a $deadlock$ state. From this point forward in the presentation we refer to the combination of the FCA and EFSM approaches as the e-FCA heuristic function.

V. RESULTS

We implemented the e-FCA heuristic function in the gnu-debugger based model checker Estes, [8], and executed it on a benchmark set consisting of programs with concurrency errors. The results show that the e-FCA heuristic reduces the total number of states generated and also decreases the total running time before error discovery compared to the FSM and EFSM distance heuristics.

We focus specifically on three classical concurrency problems in our benchmark suite: `Dining Philosophers`, `Barbershop`, and `Hyman's mutual exclusion principle`. The results presented are from a Pentium III 1.5 Ghz processor with 2 GB of RAM and are run on Estes, with a 6.1.1 version of the gnu debugger, using the m68hc11 backend simulator. We report the wall clock time for the time

TABLE II

A COMPARISON ACROSS DIFFERENT SEARCH TECHNIQUES

Name	T	M	k	Total States Generated					Time taken in Seconds				
				DFS	Rand	FSM	EFSM	e-FCA	DFS	Rand	FSM	EFSM	e-FCA
Hyman	2	3	1	6,478	15,800	10,227	7,160	3,817	3	9	4	6	1
Hyman	2	4	1	16,190	59,796	41,791	21,909	13,529	7	28	17	21	5
Hyman	2	5	1	40,471	91,947	123,743	59,951	38,745	17	42	49	56	16
D-phil	2	2	1	157,436	475,184	53,897	4,594	1,626	71	318	31	79	1
D-phil C	2	2	1	19,769	11,497	18,148	1036	415	12	10	14	77	0
D-phil	2	3	1	$*$	452,092	54,725	13,830	3,816	$*$	292	28	155	3
D-phil C	2	3	1	157,818	95,009	8,575	3,348	1,015	102	75	5	149	1
D-phil	2	4	0	$*$	999,480	186,419	36,467	13,696	$*$	730	113	27	8
D-phil C	2	4	0	548,127	173,494	42,107	10,224	3,655	299	159	31	12	3
D-phil	2	5	0	$*$	$*$	334,198	400,474	55,876	$*$	$*$	178	388	32
D-phil C	2	5	0	$*$	370,656	861,319	142,350	14,755	$*$	294	680	136	11
Barbers	3	3	1	$*$	442,285	82,333	21,465	3,298	$*$	282	41	14	2
Barbers	3	4	0	$*$	939,828	73,940	75,635	13,118	$*$	576	38	47	7
Barbers	3	5	0	$*$	$*$	378,632	388,161	66,608	$*$	$*$	237	252	38

taken to do the static analysis and for the total running time of the program before error discovery, as well as the total states generated before error discovery.

For the benchmarks, we add procedures containing nested function calls that do not affect the property being verified to the base implementation of the concurrent programs. We then randomly insert calls to these procedures throughout the programs to derive examples with varying call structures and call depths in order to test the accuracy and efficiency of the e-FCA in computing context aware distances. To measure the accuracy of the e-FCA heuristic function we compare the e-FCA to the shortest-path analysis (FSM) and the EFSM distance heuristic. We also compare it with random search and an exhaustive depth-first search (DFS). We use best-first search rather than A^* to decrease the number of states expanded before error discovery; although best-first does not guarantee a shortest error trace like the A^* search. In our experiments, the length of the error traces generated by the best-first search are comparable to that of A^*. The results of the analyses are shown in Table I, Table II and Table III. In Table I and Table II the first column (`Name`), shows the concurrent program being verified; if the program is model checked at the C-level, where a single C instruction is considered atomic, a letter C is appended to the end of the program name. Otherwise, the program is model checked at the assembly-level where a single assembly-level instruction is considered atomic. The next column (`T`) indicates the number of threads in the program, the column (`M`) shows the maximum possible call depth of the program, and the column (`k`) is the value of the bound picked for the EFSM heuristic.

The static analysis time reported in Table I is the time taken in seconds during the period after the execution of the program starts and before the model checking run begins. The FCA analysis takes negligible amount of time. The average time taken by the FCA to complete static analysis (0.65 secs) is less than the average time (1 secs) taken by the FSM, even though the FSM does not consider calling context of the program at all. As we increase the k-bound for the EFSM distance heuristic, the cost to construct the inlined graph and do a shortest path analysis for checking whether the error is in scope grows exponentially. The high overhead of static analysis with larger k-bounds forces us to pick a bound of either one or zero for the EFSM distance heuristic in order to finish static analysis within a few minutes. In spite of picking low bounds of k for the EFSM computation, the time taken by EFSM to complete static analysis is significantly higher compared to the FCA. Note that even with a bound of zero, the EFSM dynamically recreates the call trace of the program; hence, it has more context than the FSM distance heuristic.

For each model, we report the total number of states enumerated before finding the error state and total running time to find the error in Table II. The '$*$' symbol in Table II shows that after generating a million states, the error state was still not found, and at that point, the search was terminated. For the random search, the heuristic value is set to a random value and the numbers reported for the total number of states and total running time numbers are averaged over 10 model checking runs.

The e-FCA gains a significant reduction in total states generated compared to the other heuristics and search techniques as shown on the left side of Table II. In Table II, we can see that DFS, an exhaustive search is mostly ineffective in finding the error. In seven out of fourteen examples it is unable to find the error within a million states; however, sometimes DFS happens to find the error quickly by chance as seen in the `Hyman` examples. In some cases, the EFSM generates more states than the FSM distance heuristic before finding the error. Our experiments with different k-bounds show that for some programs, the improvement in error discovery by the EFSM heuristic with increasing context is not always monotonic. For such programs, the EFSM heuristic does not perform well until the context information reaches a certain threshold.

The e-FCA also obtains a significant decrease in the total running time compared to the other search techniques as shown on the right side of Table II. The state reduction achieved by the EFSM is not enough to compensate for the

TABLE III
SCALABILITY ACROSS DIFFERENT THREADS

	Depth = 2		Depth = 5		Depth = 9	
T	States	Time	States	Time	States	Time
5	814	1	7,064	5	92,434	71
9	1,070	1	7,320	6	93,230	82
11	1,196	1	7,446	11	93,356	144
15	1,448	1	7,698	12	93,608	158
18	1,641	1	7,891	12	93,801	165
20	1,767	2	8,071	13	93,927	173
25	2,086	3	8,336	15	94,246	187
30	2,401	3	8,970	18	94,561	204
40	3,040	4	9,603	22	92,500	233
51	3,736	8	10,306	30	95,896	311

high cost of static analysis causing its total running time to increase dramatically compared to the e-FCA. While computing the heuristic estimate, the EFSM faces an additional overhead cost of extracting the call trace in the run-time stack from the start of the program to its current point. The e-FCA faces the same overhead of extracting the run-time stack; however, this cost is very effectively mitigated by the significant reduction in the states generated and low cost of static analysis resulting in a substantial decrease in the total running time of the e-FCA before error discovery.

We test the scalability of the e-FCA heuristic function by instrumenting our implementation of the `barbershop` problem to allow a variable number of threads (between 5 and 51). Additionally, we implement three versions of the problem with varying maximum possible call depths of two, five, and nine. The total number of states and total running time before error discovery for these examples are presented in Table III. The first column (T) in Table III indicates the number of threads created for the particular example. From the barbershop example, it seems that the e-FCA scales to multiple threads with a high degree of nested function calls. DFS and random search do not find the error in a million states for even the smallest model. The FSM distance heuristic and EFSM distance heuristic, with small k-bounds, do not find the error in a million states for most of the models. With a slightly higher k-bound the EFSM heuristic does not finish static analysis in 1 hour for any of the examples.

VI. CONCLUSION AND FUTURE WORK

In this paper we present the FCA algorithm that computes full context aware distances for a CFG in a non-recursive program with resolved function pointers by implicitly inlining function calls. It propagates context information through call nodes, start nodes, and end nodes of the CFG and annotates the nodes in the CFG with context sensitive distances to end nodes and error locations in the forward direction. We then present a new heuristic function, e-FCA which combines the unbounded distance data computed by the FCA algorithm with the dynamic recreation of the run-time stack from the EFSM heuristic function. The e-FCA heuristic function computes more accurate heuristic estimates compared to other distance heuristic functions.

In some cases, the e-FCA heuristic function underestimates the true distance to the error locations because it does not consider the feasibility of the execution paths. Resolving the feasiblity of all execution paths is not possible statically; however, while model checking, as the variables are assigned dynamic values, we can determine the infeasible execution paths. In future work we plan on pruning these infeasible execution paths before computing the heuristic estimate to overcome the underestimation arising due to the path-insensitive computation of the e-FCA.

REFERENCES

[1] T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre, "Software verification with Blast," in *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, ser. Lecture Notes in Computer Science, T. Ball and S. Rajamani, Eds., vol. 2648, Portland, OR, May 2003, pp. 235–239.

[2] T. Ball and S. Rajamani, "The SLAM toolkit," in *13th Annual Conference on Computer Aided Verification (CAV 2001)*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Paris, France: Springer-Verlag, July 2001, pp. 260–264.

[3] J. Penix, W. Visser, C. Pasaranu, E. Engstrom, A. Larson, and N. Weininger, "Verifying time partitioning in the DEOS scheduling kernel," in *22nd International Conference on Software Engineering (ICSE00)*. Limerick, Ireland: ACM, June 2000, pp. 488–497.

[4] Robby, M. B. Dwyer, and J. Hatcliff, "Bogor: An extensible and highly-modular model checking framework," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 267–276, September 2003.

[5] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *7th International SPIN Workshop*, ser. Lecture Notes in Computer Science, K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885. Springer, August 2000, pp. 113–130. [Online]. Available: citeseer.nj.nec.com/ball00bebop.html

[6] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs." *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[7] P. Leven, T. Mehler, and S. Edelkamp, "Directed error detection in C++ with the assembly-level model checker StEAM," in *Proceedings of 11th International SPIN Workshop, Barcelona, Spain*, ser. Lecture Notes in Computer Science, vol. 2989. Springer, 2004, pp. 39–56.

[8] E. G. Mercer and M. Jones, "Model checking machine code with the GNU debugger," in *12th International SPIN Workshop*, ser. Lecture Notes in Computer Science, vol. 3639. San Francisco, USA: Springer, August 2005, pp. 251–265.

[9] C. Yang and D. Dill, "Validation with guided search of the state space," in *35th Design Automation Conference (DAC98)*, 1998, pp. 599–604. [Online]. Available: http://citeseer.nj.nec.com/yang98validation.html

[10] S. Edelkamp, A. Lluch-Lafuente, and S. Leue, "Directed explicit model checking with HSF-SPIN," in *Proc. of the 7th International SPIN Workshop*, ser. Lecture Notes in Computer Science, no. 2057. Springer-Verlag, 2001.

[11] K. Seppi, M. Jones, and P. Lamborn, "Guided model checking with a bayesian meta-heuristic." *Fundam. Inform.*, vol. 70, no. 1-2, pp. 111–126, 2006.

[12] A. Groce and W. Visser, "Model checking Java programs using structural heuristics," in *2002 ACM SIGSOFT International symposium on software testing and analysis*, 2002, pp. 12–21.

[13] S. Edelkamp and T. Mehler, "Byte code distance heuristics and trail direction for model checking Java programs," in *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, 2003, pp. 69–76.

[14] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification," in *International Conference on Software Engineering*, 2001, pp. 37–46. [Online]. Available: citeseer.ist.psu.edu/cobleigh01right.html

[15] N. Rungta and E. G. Mercer, "A Context-Sensitive Structural Heuristic for Guided Search Model Checking," in *20th IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, California, USA, November 2005, pp. 410–413.

Liveness and Boundedness of Synchronous Data Flow Graphs *

A.H. Ghamarian, M.C.W. Geilen, T. Basten, B.D. Theelen, M.R. Mousavi and S. Stuijk
Eindhoven University of Technology, Electronic Systems Group
a.h.ghamarian@tue.nl

Abstract. *Synchronous Data Flow Graphs (SDFGs) have proven to be suitable for specifying and analyzing streaming applications that run on single- or multi-processor platforms. Streaming applications essentially continue their execution indefinitely. Therefore, one of the key properties of an SDFG is liveness, i.e., whether all parts of the SDFG can run infinitely often. Another elementary requirement is whether an implementation of an SDFG is feasible using a limited amount of memory. In this paper, we study two interpretations of this property, called boundedness and strict boundedness, that were either already introduced in the SDFG literature or studied for other models. A third and new definition is introduced, namely self-timed boundedness, which is very important to SDFGs, because self-timed execution results in the maximal throughput of an SDFG. Necessary and sufficient conditions for liveness in combination with all variants of boundedness are given, as well as algorithms for checking those conditions. As a by-product, we obtain an algorithm to compute the maximal achievable throughput of an SDFG that relaxes the requirement of strong connectedness in earlier work on throughput analysis.*

1 Introduction

Synchronous Data Flow Graphs (SDFGs, see [13]), also known as weighted Marked Graphs in Petri-net theory, are used widely in modelling and analyzing data flow applications. They are often used for modelling DSP applications [3, 19] and for designing concurrent multimedia applications implemented on multi-processor systems-on-chip [17]. The model is suitable for realizing a system with predictable performance properties as several analysis techniques like throughput analysis exist [8].

An SDFG is a graph with actors as vertices and channels as edges. Actors represent basic parts of an application which need to be executed. Channels represent data dependencies between actors. Execution of an actor is designated by an actor firing. Each actor generates a fixed number of tokens when it fires. These are stored in the channels with unlimited capacities. An execution of an SDFG is a sequence of actor firings which respects data dependencies. The exact order of actor firings is not determined. Consequently, several executions exist for an SDFG. Because of the usage of SDFGs for modelling streaming applications, only those SDFGs which have executions in which all actors are fired infinitely often are of interest. This property of SDFGs is called liveness. Furthermore, only executions that require a finite amount of storage for the channels are of interest. This paper formally studies three different interpretations of this second property, all in combination with liveness.

The paper investigates two known interpretations, namely boundedness (whether there exists a bounded execution of an SDFG) and strict boundedness (whether all executions are bounded). We prove necessary and sufficient conditions guaranteeing that an SDFG is live and (strictly) bounded. For strict boundedness, these conditions follow immediately from a similar result known for Petri nets.

The natural way of executing an SDFG in which all actors fire as soon as they can fire, is called self-timed execution. This execution is important since it leads to the maximal obtainable throughput of an SDFG [19]. Because of the importance of self-timed execution of SDFGs and its applications in the context of multi-processor systems, a new notion of boundedness, namely self-timed boundedness is introduced. This notion requires that self-timed execution of SDFGs is bounded. Necessary and sufficient conditions for the liveness and self-timed boundedness of SDFGs are proved. These conditions heavily depend on the throughput of actors (average number of firings of an actor per time unit). Existing techniques for throughput calculation only work for strongly connected SDFGs [6, 8]. We propose an algorithm that determines the liveness and self-timed boundedness of an SDFG and at the same time extends throughput analysis to arbitrary SDFGs. The concept of self-timed boundedness and the results proven for this notion are the main contribution of this paper.

The rest of this paper is organized as follows. Section 2 formally introduces SDFGs to allow studying liveness and boundedness in a rigorous way. Sections 3 and 4 present results for liveness and (strict) boundedness. Section 5 identifies conditions for self-timed boundedness of SDFGs and presents an algorithm for verifying the combination of liveness and this type of boundedness. Section 6 discusses related work, while Section 7 summarizes the conclusions. Proofs are omitted and can be found in [9].

2 Synchronous Data Flow Graphs

2.1 Basic Definitions

This section formally defines SDFGs and some of their basic properties. Let $\mathbb{N}_0 = \{0, 1, \ldots\}$ (and $\mathbb{N} = \mathbb{N}_0 \setminus$

*This work was supported by the Dutch Science Foundation NWO, project 612.064.206, PROMES, and the EU, project IST-004042, Betsy.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

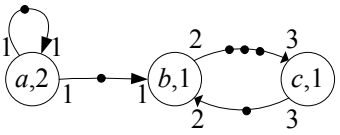


Figure 1. An example timed SDFG G_{ex}.

$\{0\}$) denote the (positive) natural numbers. The following definition captures the structure of an SDFG.

Definition 1 *[Synchronous Data Flow Graph (SDFG)] An SDFG is a pair (A, C), where A denotes the set of actors and $C \subseteq A^2 \times \mathbb{N}^2$ the set of channels. Each $(s, d, p, c) \in C$ denotes that actor d depends on actor s, where p and c are the production and consumption rates of tokens of s and d, respectively. The predecessors of a in $\mathrm{Pred}(a) = \{s \in A \mid (s, a, p, c) \in C\}$ are those actors on which a depends. The channels between a and its predecessors are referred to as the input channels of a, denoted by $\mathrm{IC}(a)$. Similarly, the successors of a in $\mathrm{Succ}(a) = \{d \in A \mid (a, d, p, c) \in C\}$ are those actors that depend on a. The output channels (channels between a and its successors) of a are denoted by $\mathrm{OC}(a)$. We call a channel from an actor a to itself a* self-loop channel. *We denote the set of self-loop channels of an actor a by $\mathrm{SLC}(a) = \mathrm{IC}(a) \cap \mathrm{OC}(a)$. An SDFG in which all production and consumption rates are one is called a* Homogeneous *SDFG (HSDFG).*

Figure 1 shows a simple example of an SDFG. Actors are labeled with their names and execution times (introduced later). Channels are labeled with production and consumption rates. The black dots are tokens. To capture the execution of an SDFG, we define the channel state of an SDFG as the distribution of tokens over its channels.

Definition 2 *[Channel State] A* channel state *of an SDFG (A, C) is a function $S : C \to \mathbb{N}_0$ that returns the number of tokens stored in each channel. Each SDFG has an initial channel state S_0 denoting the number of tokens that are initially stored in the channels.*

An execution of an SDFG is defined based on the firings of its actors, which may lead to changes in the channel state.

Definition 3 *[Firing] Let $a \in A$ be an actor of an SDFG (A, C). Actor a is said to be enabled in channel state S in case $S(e) \geq c$ for all input channels $e = (s, a, p, c)$ in $\mathrm{IC}(a)$. If a is enabled in S_i and it fires, the resulting channel state S_{i+1} is defined by $S_{i+1}(e) = S_i(e) - c$ for each input channel $e = (s, a, p, c)$ in $\mathrm{IC}(a) \backslash \mathrm{SLC}(a)$, $S_{i+1}(e) = S_i(e) + p$ for each output channel $e = (a, d, p, c)$ in $\mathrm{OC}(a) \backslash \mathrm{SLC}(a)$, $S_{i+1}(e) = S_i(e) + p - c$ for each self-loop channel $e = (a, a, p, c) \in \mathrm{SLC}(a)$, and $S_{i+1}(e) = S_i(e)$ for all channels $e \notin \mathrm{IC}(a) \cup \mathrm{OC}(a)$.*

Definition 4 *[Execution and Maximal Execution] Let S_0 denote the initial channel state of an SDFG (A, C). An* execution *σ of (A, C) is a (finite or infinite) sequence of channel states $S_0, S_1 \ldots$ such that S_{i+1} is the result of firing an enabled actor in S_i for all $i \geq 0$. An execution is maximal if and only if it is finite with no actors enabled in the final channel state, or if it is infinite.*

Not all SDFGs are considered to be useful in practice. One normally seeks a system that is deadlock-free or live.

Definition 5 *[Deadlock and Liveness] An SDFG has a* deadlock *if and only if it has a maximal execution of finite length. An SDFG is* live *if and only if it has an execution in which all actors fire infinitely often.*

It is known [11] that the execution of an SDFG is determinate, which means that the order of execution does not affect the states that can eventually be reached. Thus, if one execution of an SDFG deadlocks, then all executions deadlock. The example SDFG G_{ex} is live.

2.2 Timed SDFGs

For performance analysis of streaming applications, an SDFG is often extended with time.

Definition 6 *[Execution Time] An* execution time *models the execution duration of actors for SDFGs. In an SDFG (A, C), the execution time is a function $E : A \to \mathbb{Q}_0^+ \cup \{\infty\}$ that assigns to each actor the amount of time it takes to fire, where $\mathbb{Q}_0^+ \cup \{\infty\}$ is the set of positive rational numbers plus 0 and ∞. For $a \in A$, $E(a)$ is referred to as the execution time of a.*

Definition 7 *[Timed SDFG] A* timed SDFG *is a triple (A, C, E) denoting an SDFG (A, C) with execution time E.*

The infinite execution times are used lateron to model deadlocks. Normally, SDFGs do not have infinite actor execution times.

Notice that actor firings in a timed SDFG are not atomic. Firing an actor now takes time. To define the state of a timed SDFG, we assume that all changes in the number of tokens on all channels of an actor happen at the end of its firing.

Definition 8 *[Timed State] A state of a timed SDFG (A, C, E) is a pair (S, τ), where S is a channel state and $\tau \in \mathbb{Q}_0^+$ is the accumulated time. The initial state of (A, C, E) is given by the initial channel state S_0 and the start time of the system $\tau_0 = 0$.*

Definition 9 *[Timed Execution] An execution of a timed SDFG (A, C, E) is a sequence of timed states $(S_0, \tau_0), (S_1, \tau_1), \ldots$, where $\tau_{i+1} \geq \tau_i$. Each two consecutive states (S_{i+1}, τ_{i+1}) and (S_i, τ_i) are the same except that an actor a which started its firing at $\tau_{i+1} - E(a)$ finishes its firing at τ_{i+1}. S_{i+1} is related to S_i in precisely the same way as defined in Definition 3.*

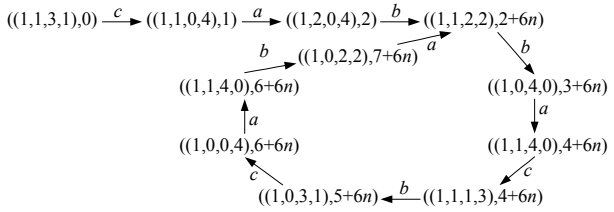


Figure 2. Self-timed execution of G_{ex}.

We denote the number of completed firings of an actor $a \in A$ which occurred up to time τ by $F_{a,\tau}$.

Among all timed executions there are some of special interest. A timed execution for which the firing of an actor always starts as soon as possible is called a *self-timed execution*. Self-timed executions are important in the context of performance analysis because they imply obtaining the maximal attainable throughput [19].

Definition 10 *[Self-timed Execution] A timed execution is called* self-timed *if and only if it is maximal and all actors start their firing as soon as they are enabled.*

If two or more actors complete their firing at some point in time in a self-timed execution, the order of their appearance in the execution is not determined. In other words, any permutation of such actor firings results in a self-timed execution. Thus, the number of self-timed executions is larger than one in such cases. Note that in all self-timed executions the start and end times of firings of all actors are equal. Also the channels states after completion of all actor firings that can complete at a certain point in time are the same in all self-timed executions.

Figure 2 illustrates a self-timed execution of the example SDFG G_{ex} of Figure 1. The state contains a channel component with the distribution of tokens over the channels a-a, a-b, b-c, c-b, respectively, and a time component. In the depicted cycle, the time component is denoted symbolically to emphasize that the behavior repeats itself every six time units, after some initial transient phase.

2.3 Structural Properties

The directed graph of an SDFG has some structural properties that are relevant for deciding boundedness. This paper assumes connected SDFGs for which the directed graph consists of *one* component. SDFGs consisting of multiple components can be considered as a set of single-component SDFGs, which can be analyzed separately.

A well known stronger form of connectivity is given by the following two definitions.

Definition 11 *[Path and Cycle] A directed path p is a sequence of actors $a_1, a_2 \ldots a_l$ such that $a_{i+1} \in \mathrm{Succ}(a_i)$ for all $1 \leq i < l$. Path p is simple iff $a_i \neq a_j$ for all $i \neq j$. If $a_1 = a_l$ and $l \geq 2$, then p is said to be a* cycle.

Definition 12 *[Strongly Connected SDFG] An SDFG is* strongly connected *iff there exists a directed path from any actor to any other actor. Any subgraph of an SDFG which is strongly connected is called a strongly connected component (SCC, for short). An SCC κ is maximal iff there is no SCC κ' where κ is a strict subgraph of κ'.*

Another structural property of SDFGs concerns the correspondence between production and consumption rates.

Definition 13 *[Consistency and Balance Equations] A repetition vector for an SDFG (A, C) is a function $\gamma : A \to \mathbb{N}_0$ such that for every $(s, d, p, c) \in C$, the equation $p\gamma(s) = c\gamma(d)$ holds. These equations are called* balance equations. *Repetition vector γ is called non-trivial iff $\gamma(a) > 0$ for all $a \in A$. If a non-trivial repetition vector exists, the SDFG is called* consistent. *The smallest non-trivial repetition vector of a consistent SDFG is referred to as* the *repetition vector.*

Note that the definitions in this subsection carry over to timed SDFGs in a straightforward way. Timed SDFG G_{ex} is consistent with repetition vector $(a \mapsto 3, b \mapsto 3, c \mapsto 2)$.

2.4 Throughput of Timed SDFGs

In this section the throughput of timed SDFGs is defined, and the relation between the execution of an SDFG and its throughput is explained.

Definition 14 *[Throughput] The* throughput $Th(a)$ *of an actor a for a self-timed execution of a timed SDFG (A, C, E) is defined as the average number of firings of a per time unit. Formally,*

$$Th(a) = \lim_{\tau \to \infty} \frac{F_{a,\tau}}{\tau}.$$

If $G = (A, C, E)$ is consistent, then its throughput is defined as

$$Th(G) = \min_{a \in A} \frac{Th(a)}{\gamma(a)},$$

where γ is the repetition vector of (A, C, E). That is, the throughput of G is the minimal actor throughput normalized by the repetition vector.

We define the *local* throughput of an actor as the throughput of that actor in a self-timed execution where non-self-loop input channels are removed; in other words, the throughput of an actor when it does not need to wait for data from other actors.

Definition 15 *[Local Throughput] The* local throughput $LTh(a)$ *of an actor a for a self-timed execution of a timed SDFG (A, C, E) is defined as*

$$LTh(a) = \begin{cases} 0, & \text{if there is a } ch = (a, a, p, c) \text{ in } \mathrm{SLC}(a) \\ & \text{such that } p < c \text{ or } S_0(ch) < c \\ \min\limits_{ch=(a,a,r,r) \in \mathrm{SLC}(a)} \lfloor S_0(ch)/r \rfloor / E(a), & \text{otherwise.} \end{cases}$$

If an actor has a self-loop channel with a lower production rate than consumption rate or insufficient tokens for an initial firing, its local throughput is zero, i.e., it deadlocks at some point in time. Otherwise, the local throughput is determined by the self-loop channels with equal production and consumption rates. If there are no such channels, i.e., there are no self-loop channels or all self-loop channels have a higher production than consumption rate, local throughput is by definition infinite.

In a self-timed execution of a timed SDFG, there is always a time τ_p after which only a repetitive pattern of actor firings occurs (when ignoring the order among actor firing completions occurring at the same moment in time) [8, 1]. The self-timed execution from the beginning up to time τ_p is called the transient phase, and thereafter is addressed as the periodic phase. Figure 2 illustrates this fact. Thus, the throughput of an arbitrary actor a in the self-timed execution can be calculated by counting the number of occurrences of firings of a in one period divided by the amount of time that the period takes. The firings of a in one period can be spread over the period, but the number of firings of one actor in one period is always fixed.

Consider again SDFG G_{ex} of Figure 1. The local throughput of actor a is $\frac{1}{2}$, whereas it is ∞ for b and c. The throughput of the three actors equals $\frac{3}{6} = \frac{1}{2}$, $\frac{3}{6} = \frac{1}{2}$, and $\frac{2}{6} = \frac{1}{3}$, respectively. The graph throughput $Th(G_{ex})$ is determined by actor a (with repetition-vector entry 3) and is equal to $(\frac{3}{6})/3 = \frac{1}{6}$. This illustrates that the periodic behavior of the graph as a whole needs 6 time units per period.

2.5 Boundedness Definitions

Different useful notions of boundedness can be defined for SDFGs. To enable identifying these forms, we first define boundedness for a given execution.

Definition 16 *[Bounded Channel and Bounded Execution] Let $\sigma = S_0, S_1, \ldots$ be an execution of an SDFG (A, C). We call a channel ch bounded under σ iff there exists some $B \in \mathbb{N}$ such that $S_i(ch) \leq B$ for all $i \geq 0$. If all channels of the SDFG are bounded under σ then σ is bounded.*

Definition 16 carries over to timed executions in a straightforward way. Now, we give a definition for the boundedness of an SDFG which intuitively means that it can be implemented using a finite amount of memory.

Definition 17 *[Bounded SDFG] A (timed) SDFG is called* bounded *iff there exists a bounded maximal execution. It is unbounded otherwise.*

A stronger form of boundedness is *strict boundedness*.

Definition 18 *[Strictly Bounded Channel and Strictly Bounded SDFG] A channel is* strictly bounded *iff it is bounded under all executions. A (timed) SDFG is called strictly bounded iff all of its channels are strictly bounded.*

Note that any *strictly bounded* SDFG is also bounded. We finally define another form of boundedness, which only considers self-timed executions of timed SDFGs.

Definition 19 *[Self-timed Bounded SDFG] A timed SDFG is* self-timed bounded *iff all self-timed executions are bounded. A channel in a timed SDFG is self-timed bounded iff it is bounded in all self-timed executions.*

All self-timed bounded SDFGs are bounded but not necessarily strictly bounded. Running example G_{ex} is not strictly bounded because a can be fired indefinitely without firing b and c. However, it is self-timed bounded, as Figure 2 illustrates. It is not difficult to construct bounded SDFGs that are not self-timed bounded. If the execution times of actors b and c in G_{ex} are changed to 3, for example, then the SDFG remains bounded but it is no longer self-timed bounded. These examples show that the notion of self-timed boundedness does not coincide with other notions of boundedness. Given the importance of self-timed execution, it is worth investigating this notion in some detail.

3 Boundedness

In this section, we study necessary and sufficient conditions under which an SDFG is live and bounded.

Theorem 20 *A live SDFG $G = (A, C)$ is bounded iff it is consistent.*

Theorem 20 states the consistency of an SDFG as a necessary and sufficient condition for boundedness of live SDFGs. If a subgraph of an SDFG deadlocks (which means that the SDFG is not live) then the consistency of an SDFG is not sufficient for boundedness. For example, consider G_{ex} of Figure 1 without the initial token in the c-b channel. Execution times may be ignored. The resulting SDFG is consistent but not bounded. The SCC of the graph that consists of actors b and c deadlocks after the first firing of both actors. However, actor a can continue its firing, which leads to an unbounded channel between a and b.

Proposition 21 *[20] A strongly connected SDFG is live iff it is deadlock-free.*

The definition of liveness states that a live SDFG has an execution in which all actors fire infinitely often. If a live SDFG is strongly connected, then all actors fire infinitely often in *all maximal* executions.

Lemma 22 *If one SCC in an SDFG G deadlocks then either G deadlocks or it is unbounded.*

This lemma implies that a deadlock-free and bounded SDFG is live.

Corollary 23 *An SDFG is live and bounded iff it is deadlock-free and bounded.*

The following theorem follows from Theorem 20, Proposition 21, Lemma 22, and Corollary 23.

Theorem 24 *An SDFG is live and bounded iff it is consistent and all its SCCs are deadlock-free.*

The example SDFG G_{ex} is live and bounded because it is consistent and all its SCCs are deadlock-free.

Next, we give an algorithm to check liveness and boundedness of an SDFG.

Algorithm *isLive&Bounded*(G)
Input: A connected (timed) SDFG G
Output: "live and bounded" or "either deadlock or unbounded"
1. **if** G is inconsistent
2. **then return** "either deadlock or unbounded"
3. **for** each maximal SCC S in G
4. **do if** S deadlocks
5. **then return** "either deadlock or unbounded"
6. **return** "live and bounded"

Consistency of SDFGs can be verified efficiently as explained in [3]. Maximal SCCs of a graph can also be computed efficiently [5]. Algorithms for detecting deadlock for consistent strongly connected SDFGs that are efficient in practice are given in [12, 8].

4 Strict Boundedness

This section identifies sufficient and necessary conditions for the liveness and strict boundedness of an SDFG.

Theorem 25 *[20, Theorem 4.11] A live (timed) SDFG is strictly bounded iff it is consistent and strongly connected.*

This theorem in combination with Proposition 21 implies the following theorem.

Theorem 26 *An SDFG is live and strictly bounded iff it is deadlock-free, consistent and strongly connected.*

So the algorithm for checking liveness and strict boundedness first checks whether the SDFG is strongly connected and consistent, and then whether it is deadlock-free using the algorithms from [5, 3, 8, 12]. The example of Figure 1 is not strictly bounded because it is not strongly connected.

5 Self-timed Boundedness

In this section, we investigate the liveness and self-timed boundedness of timed SDFGs. A self-timed execution of a live and self-timed bounded SDFG uses a finite amount of memory and all actors fire infinitely often in such an execution. Necessary and sufficient conditions for liveness and self-timed boundedness are given, and an algorithm for checking these conditions.

5.1 Some Basic Properties

Self-timed boundedness has a strong relationship with the throughput of an SDFG. In this subsection, some properties for the throughput as well as the relation between boundedness and throughput of timed SDFGs are given.

The throughput of an actor is only determined by the throughput of its predecessors and its local throughput.

Lemma 27 *The throughput of an actor $b \in A$ of a timed SDFG $G = (A, C, E)$ satisfies the equation*

$$Th(b) = \min\{\min_{(a,b,p,c)\in \mathrm{IC}(b)\setminus \mathrm{SLC}(b)} \frac{p}{c} Th(a), LTh(b)\}. \quad (1)$$

The throughput of actor b of G_{ex}, for example, is $\frac{1}{2}$, because its predecessor a has that throughput, the rates of channel a-b are 1 and its local throughput is ∞.

Corollary 28 *If actors $a, b \in A$ of an SDFG G are connected by a channel (a, b, p, c) then $Th(b) \leq (p/c)\, Th(a)$.*

After having illustrated the factors that are involved in calculating the throughput of an actor, we now show that the only case that a channel is not self-timed bounded, is when the production of tokens into one channel is larger than the consumption of tokens out of that channel.

Lemma 29 *SDFG (A, C, E) is self-timed bounded iff $Th(b) \geq (p/c)\, Th(a)$ for every channel $(a, b, p, c) \in C$.*

The next proposition gives necessary and sufficient conditions for self-timed boundedness of a live strongly connected SDFG.

Proposition 30 *A live and strongly connected SDFG G is self-timed bounded iff it is consistent.*

Lemmas 31 and 32 and Proposition 33 prove some useful properties about the relation between the throughput of various actors. Lemma 31, which follows immediately from Corollary 28 and Lemma 29, shows the relation between producer and consumer actors of an arbitrary self-timed bounded channel. Lemma 32 shows the relation between the actor throughputs for any two actors in an SCC of an SDFG. Proposition 33 gives the relation between the throughput of two arbitrary actors in consistent self-timed bounded or strongly connected SDFGs.

Lemma 31 *If a channel (a, b, p, c) connecting a and b is self-timed bounded then $Th(b) = (p/c)\, Th(a)$.*

Lemma 32 *If a and b are two actors of an SCC of a consistent SDFG with repetition vector γ, then $Th(a)/\gamma(a) = Th(b)/\gamma(b)$.*

Proposition 33 *If a and b are two actors of a consistent self-timed bounded or strongly connected SDFG G with repetition vector γ then $Th(a)/\gamma(a) = Th(b)/\gamma(b)$.*

This proposition shows that for consistent self-timed bounded or strongly connected SDFGs the throughput as defined in Definition 14 can be calculated via an arbitrary actor without explicitly computing the minimum.

5.2 Reduction to an HSDFG

In this section, we propose a method for reducing a consistent SDFG G to an HSDFG G_H which preserves (non-)liveness and self-timed (un)boundedness of G. In G_H, every actor has a self-loop channel with one initial token, rates of all channels are one (i.e., it is an HSDFG), and, ignoring self-loops, it is acyclic. Because of these simple properties, we use the reduced graph for verifying the liveness and self-timed boundedness of the original SDFG. The reduction also preserves throughput which means our algorithm also provides the throughput of the original SDFG G.

The reduction uses the notion of local throughput of an SCC of an SDFG, and it is illustrated in Figure 3 which provides the reduced graph for the running example.

Definition 34 *[Local Throughput of an SCC] The* local throughput $LTh(\kappa)$ *of an SCC* $\kappa = (A_\kappa, C_\kappa, E_\kappa)$ *in a consistent SDFG* $G = (A, C, E)$ *with repetition vector* γ *is defined as the actor throughput of an arbitrary actor* $a \in A_\kappa$ *when all input channels from* $A \backslash A_\kappa$ *to* A_κ *are removed, divided by* $\gamma(a)$.

Lemma 32 implies that this definition is sound.

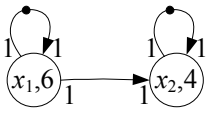


Figure 3. The reduced HSDFG for G_{ex}.

Definition 35 *[Reduced Graph] Let a consistent SDFG* $G = (A, C, E)$ *contain* n *maximal SCCs* $\kappa_1 = (A_{\kappa_1}, C_{\kappa_1}, E_{\kappa_1}), \ldots, \kappa_n = (A_{\kappa_n}, C_{\kappa_n}, E_{\kappa_n})$. *Suppose* γ *is the repetition vector of* G. *We define the reduced SDFG* $G_H = (A_H, C_H, E_H)$ *as follows:* $A_H = \{x_i | 1 \leq i \leq n\}$ *(which means one actor for each maximal SCC in* G*);* C_H *contains a channel* $(x_i, x_j, p\gamma(a), c\gamma(b))$ *for every channel* $(a, b, p, c) \in C$ *where* $a \in A_{\kappa_i}$, $b \in A_{\kappa_j}$, $i \neq j$; C_H *also contains self-loop channels* $(x_i, x_i, 1, 1)$ *for every actor; the execution time* $E_H(x_i)$ *equals* $1/LTh(\kappa_i)$ *if* κ_i *does not deadlock and* ∞ *if it does. According to the balance equations we know that for each channel in the original graph* (a, b, p, c), $p\gamma(a) = c\gamma(b)$. *Thus, the production and consumption rates for every channel in* C_H *are equal. Therefore, we can simplify the reduced* G *by setting all rates of all channels in* C_H *to one. Consequently, we obtain an HSDFG as the result. Finally, every self-loop channel in* G_H *contains one initial token, and all the other channels are empty.*

Since the HSDFG resulting from the reduction is acyclic when ignoring self-loops, the preservation of throughput, (non-)liveness and self-timed (un-)boundedness that we are aiming at, is independent of the number of initial tokens on the non-self-loop channels. Hence, we choose to leave those channels empty.

Consider the reduced graph shown in Figure 3. The original graph G_{ex} has two maximal strongly connected components, containing actor a, and actors b and c, respectively. These SCCs are reduced to actors x_1 and x_2. Since actor a has throughput $\frac{1}{2}$ and repetition-vector entry 3, the execution time of x_1 is set to 6, illustrating that 3 firings of a take 6 time units. Considering the other SCC in isolation, it can be verified that one period of this SCC containing 3 firings of b and 2 of c consists of 4 time units. Given the repetition vector of G_{ex} and Definition 34, this gives a local throughput of $\frac{1}{4}$ and an execution time of 4 for x_2.

The following proposition shows the relation between the throughput of actors in a maximal SCC of an SDFG and the throughput of the actor corresponding to that SCC in the reduced SDFG.

Proposition 36 *Let* G_H *be the reduced SDFG of a consistent timed SDFG* G *with repetition vector* γ. *If a maximal SCC* $\kappa = (A_\kappa, C_\kappa, E_\kappa)$ *in* G *is replaced by actor* x *in* G_H, *then for any* $a \in A_\kappa$, $Th(a) = \gamma(a)\, Th(x)$.

It is easy to verify that Proposition 36 holds for the running example. Consider for instance actor x_2 of the reduced graph. Its throughput in the reduced graph is fully determined by the throughput of x_1 and becomes therefore $\frac{1}{6}$. Proposition 36 states that $Th(b) = 3(\frac{1}{6}) = \frac{1}{2}$ and $Th(c) = 2(\frac{1}{6}) = \frac{1}{3}$, which corresponds to the throughput values for b and c computed at the end of Section 2.4.

The next corollary follows from the definition of throughput, the observation that all repetition-vector entries of an HSDFG are always one, and Propositions 33 and 36.

Corollary 37 *The throughput of a consistent self-timed bounded SDFG is equal to the throughput of its reduced graph.*

The reduction also preserves self-timed (un-)boundedness.

Theorem 38 *A consistent timed SDFG is self-timed bounded iff its reduced graph is self-timed bounded.*

Proposition 36 implies that non-zero throughput (i.e., (non-)liveness) is preserved.

Corollary 39 *A consistent timed SDFG is live iff its reduced graph is live.*

5.3 Verifying Self-timed Boundedness

This section introduces an algorithm that determines whether an SDFG is live and self-timed bounded. The following theorem follows from the results obtained so far.

Theorem 40 *A timed SDFG* G *is live and self-timed bounded iff* $isLive\&SelftimedBounded(G)$ *returns "yes".*

Algorithm *isLive&SelftimedBounded*(G=(A, C, E))
Input: A connected timed SDFG G
Output: "yes, $Th(G)$" if self-timed bounded and live, "no" otherwise

1. **if** not *isLive&Bounded*(G)
2. **then return** "no"
3. $G_H = (A_H, C_H, E_H) \leftarrow$reduce($G$)
4. $AL[1..|A_H|] \leftarrow$topologicalSort(G_H)
5. **if** $|A_H| = 1$
6. **then return** "yes, $\frac{1}{E_H(AL[1])}$"
7. **for** $i \leftarrow 1$ **to** $|A_H|$
8. **do** $AL[i].Th \leftarrow \frac{1}{E_H(AL[i])}$
9. **if** $\mathrm{Pred}(AL[i]) = \{AL[i]\}$ **and** $AL[i].Th = \infty$
10. **then return** "no"
11. $maxPTh \leftarrow 0$
12. **for** each $j \in \mathrm{Pred}(AL[i]) \backslash \{AL[i]\}$
13. **do** $AL[i].Th \leftarrow \min(AL[i].Th, AL[j].Th)$
14. $maxPTh \leftarrow \max(maxPTh, AL[j].Th)$
15. **if** $maxPTh > AL[i].Th$
16. **then return** "no"
17. **return** "yes, $AL[1].Th$"

The algorithm works in two steps. The first step checks the liveness and boundedness (as defined by Definition 17) of the graph by calling algorithm *isLive&Bounded* (lines 1 and 2). If the graph is not live and bounded, it cannot be live and self-timed bounded. The second step concerns determining whether the reduced HSDFG is self-timed bounded (lines 3 to 17).

If *isLive&Bounded* returns "yes", we know that the SDFG is consistent. Then, line 3 of the algorithm reduces the SDFG according to Definition 35 and stores the result in G_H. Note that the reduction requires throughput calculations for all SCCs. For efficiency reasons, these throughput calculations can be delayed till the algorithm really needs this information. Calculations may then be avoided if the algorithm returns "no" early. We have not made this explicit in the algorithm. Since G is at this point known to be live and consistent, by Corollary 39, also G_H is live. It remains to determine self-timed (un-)boundedness.

Ignoring self-loops, G_H is acyclic. Line 4 topologically sorts the actors of G_H, and stores them in array AL, so that the predecessors of an actor $AL[i]$ are only among the $AL[j]$ for $j \leq i$. If G_H contains only one actor, then G is strongly connected, and hence, by Proposition 30, self-timed bounded, and the algorithm terminates. Based on Corollary 37, it returns the local throughput of the only actor of G_H as the throughput of G. Note that every actor in a reduced graph has a self-loop channel with one token on it, so this value is equal to $1/E_H(AL[1])$. Also note that $E_H(AL[1])$, and $E_H(AL[i])$ in general, may be 0. In this case, we assume that $1/E_H(AL[i])$ is equal to ∞.

Each iteration of the loop of lines 7 to 16 starts by calculating the local throughput of each actor $AL[i]$, $1 \leq i \leq |A_H|$, storing the result in $AL[i].Th$. In case of detecting a source actor (an actor without any input channel except its self-loop channel) with an infinite throughput, the algorithm returns "no", because this implies that its output channels are unbounded. The loop continues by setting $maxPTh$ to zero. This variable is a temporary variable for storing the maximum throughput of the predecessors of actor $AL[i]$ in iteration i. In the loop of lines 12 to 14, the minimum between the local throughput of actor $AL[i]$ and the minimum throughput of its predecessors is assigned to $AL[i].Th$. This value, according to Lemma 27, is the throughput of the actor $AL[i]$. Note that since the actors are topologically sorted in AL, the throughput of all predecessors has already been calculated. The maximum throughput of the predecessors of actor $AL[i]$ is assigned to $maxPTh$.

The test of line 15 checks whether the maximum throughput of predecessors of actor $AL[i]$ (excluding $AL[i]$) is greater than the throughput of actor $AL[i]$ itself. In case it is, according to Lemma 29 at least one channel connecting a predecessor of actor $AL[i]$ to $AL[i]$ is unbounded.

If the algorithm reaches line 17, then no unbounded channel has been detected, and the graph is live and self-timed bounded. According to Corollary 37 and the fact that the reduced SDFG is an HSDFG with all repetition-vector entries one, the value of $AL[i].Th$ for all actors $AL[i] \in A_H$ is equal to the throughput of G. The algorithm returns $AL[1].Th$. The emphasis of algorithm *isLive&SelftimedBounded* is on verifying liveness and self-timed boundedness of an SDFG, so it returns as soon as it detects that the graph is not live or not self-timed bounded. It can be easily adapted to compute the throughput for SDFGs which are not self-timed bounded as well.

6 Related Work

There are interesting similarities between SDFGs and Petri nets. In particular, there is a straightforward translation from SDFGs to a subclass of Petri nets, called weighted Marked Graphs and vice versa, where actors are transitions, and channels are places. Marked Graphs, also called T-Graphs are known to be the subclass of Petri nets that is most amenable to rigorous analysis. Thus, it makes sense to compare the results obtained in this paper with the corresponding results in the literature concerning Petri nets. We studied liveness in combination with three different definitions of boundedness (Definitions 17, 18 and 19) for (timed) SDFGs.

We do not know of any related results for boundedness as defined by Definition 17. The only result we know for this type of boundedness is in [16] which only introduces it without providing necessary and sufficient conditions, as we do.

For strict boundedness in the sense of Definition 18, the problem has been studied from different viewpoints in the Petri-net literature (see for an overview [7, 15]). In particular, [20] gives necessary and sufficient conditions for strict boundedness of live weighted Marked Graphs (our Theorem 25). Strict boundedness is also the only kind of boundedness which has been investigated formally in the literature on SDFGs themselves; Karp and Miller in their seminal paper [11] introduced computation graphs, which are slightly more general than SDFGs. They proved necessary and sufficient conditions for liveness and strict boundedness

in their model. Their results as well as those in [20] correspond to those presented in this paper.

Our third definition of boundedness, self-timed boundedness (see Definition 19) is defined on timed SDFGs. Therefore, we need to compare it with time-enabled Petri nets. Petri nets have been extended with quantitative time in different ways, by adding timing information to places, transitions and/or tokens (see [4] for a survey). The timed Petri net model that comes closest to timed SDFGs is the "time Petri net" model originally defined by [14]. This extension of Petri nets associates a duration (delay) and a deadline to transitions. We are not aware of any study of the self-timed boundedness problem for the subclass of time Marked Graphs. In [18], the liveness and strict boundedness problem for time Petri nets is studied but only some sufficient conditions are given. These conditions guarantee that once a time Petri net satisfies certain syntactic constraints, it is live and strictly bounded if the underlying untimed Petri net is live and strictly bounded. Unfortunately, the results of [18] cannot be applied in our setting since the syntactic constraints require the absence of either duration or deadline both of which are necessary for translation of timed SDFGs to time Petri nets. [10] proves a general undecidability result for strict boundedness of time Petri net of [14]. However, in [2], two sufficient conditions are given for strict boundedness of time Petri nets. We are not aware of any result about self-timed boundedness as defined in Definition 19. To the best of our knowledge, both the concept and the derived results are novel.

7 Conclusions

We have studied the liveness and boundedness of Synchronous Data Flow Graphs, which are also known as weighted Marked Graphs in the Petri-net literature. Liveness and boundedness is a prerequisite of any meaningful SDFG model of a streaming multi-media application. Two known notions of boundedness, namely boundedness and strict boundedness, have been studied rigorously, and in particular necessary and sufficient conditions for liveness in combination with these two types of boundedness have been given. For strict boundedness, these conditions were already known from the Petri-net literature. Furthermore, a new notion, self-timed boundedness, was introduced. Self-timed boundedness checks whether self-timed execution of an SDFG is bounded. A self-timed execution yields the maximum throughput for an SDFG. Necessary and sufficient conditions for self-timed boundedness and liveness have been proven. An algorithm for checking these conditions was presented. Besides, existing throughput analysis techniques, which are only valid for strongly connected graphs, are extended to arbitrary consistent SDFGs.

References

[1] F. Baccelli, G. Cohen, G. Olsder, and J.-P. Quadrat. *Synchronization and linearity: an algebra for discrete event systems*. Wiley, 1992.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[3] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal on VLSI Signal Process. Syst.*, 21(2):151–166, 1999.

[4] F. D. Bowden. A brief survey and synthesis of the roles of time in Petri nets. *Mathematical and Computer Modelling*, 31(10):55–68, 2000.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[6] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Trans. on Design Automation of Electronic Systems*, 9(4):385–418, 2004.

[7] J. Esparza. Decidability and complexity of Petri net problems - an introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428. Springer-Verlag, 1998.

[8] A. H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *ACSD, Proc.*, pages 25–34. IEEE, 2006.

[9] A. H. Ghamarian, M. C. W. Geilen, T. Basten, B. Theelen, M. M. R., and S. Stuijk. Liveness and boundedness of synchronous data flow graphs. Tech. report ESR-2006-04, TU Eindhoven, http://www.es.ele.tue.nl/esreports/, 2006.

[10] N. D. Jones, L. H. Landweber, and Y. E. Lien. Complexity of some problems in Petri nets. *Theoretical Computer Science*, 4(3):277–299, 1977.

[11] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[12] E. Lee. *A coupled hardware and software architecture for programmable digiral signal processors*. PhD thesis, University of California, Berkeley, 1986.

[13] E. Lee and D. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[14] P. M. Merlin. *A Study of Recoverability of Processes*. PhD thesis, Department of Information and Computer Science, University of California at Irvine, 1975.

[15] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[16] T. M. Parks. *Bounded Scheduling for Process Networks*. PhD thesis, 1995.

[17] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level timing models for guaranteed performance in multiprocessor networks-on-chip. In *CASES, Proc.*, pages 63–72. ACM, 2003.

[18] L. Popova-Zeugmann. On liveness and boundedness in time Petri nets. In *Proceedings of the Workshop on Concurrency, Specification and Programming (CS&P'95)*, pages 136–145, 1995.

[19] S. Sriram and S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc, New York, NY, USA, 2000.

[20] E. Teruel, P. Chrzastowski, J. M. Colom, and M. Silva. On weighted T-systems. In Jensen, K., editor, *13th Internationald Conference on Application and Theory of Petri Nets 1992, Sheffield, UK*, volume 616 of Lecture Notes in Computer Science, pages 348–367. Springer-Verlag, 1992.

Model Checking Data-Dependent Real-Time Properties of the European Train Control System

Johannes Faber and Roland Meyer
University of Oldenburg, Germany
{johannes.faber|roland.meyer}@uni-oldenburg.de

I. Introduction

The behavior of embedded hardware and software systems is determined by at least three dimensions: control flow, data aspects, and real-time requirements. To specify the different dimensions of a system with the best-suited techniques, the formal language CSP-OZ-DC [1] integrates Communicating Sequential Processes (CSP) [2], Object-Z (OZ) [3], and Duration Calculus (DC) [4] into a declarative formalism equipped with a unified and compositional semantics. In this paper, we provide evidence that CSP-OZ-DC is a convenient language for modeling systems of industrial relevance. To this end, we examine the emergency message handling in the European Train Control System (ETCS) [5] as a case study with uninterpreted constants and infinite data domains. We automatically verify that our model ensures real-time safety properties, which crucially depend on the system's data handling.

Related work on ETCS case studies focuses on stochastic examinations of the communication reliability [6], [7]. The components' data aspects are neglected, though.

II. A CSP-OZ-DC Model of the ETCS

In this section, we introduce the case study and present its CSP-OZ-DC model in more detail.

The ETCS [5] aims at replacing national train control systems with the goal of ensuring cross-border interoperability and improving railway safety as well as track utilization. In the final ETCS implementation level, trains communicate over a radio connection with *radio block centers* (RBCs). Controlling the traffic in well-defined areas, the RBCs grant movement authorities to trains. In order to increase the traffic density, movement authorities are given up to a position—called *limit of authority* (LOA)—closely behind the preceding train. In case of an accident, the train control system has to stop all trains safely. Focusing on the emergency message handling, we ensure in our case study that the trains never collide.

In our CSP-OZ-DC model the case study's components are specified in an object-oriented way using classes. To give an idea of both the specification language and the case study, we introduce the class *Train* in more detail. Every CSP-OZ-DC class comprises an **interface** part defining typed channels that can be used for the inter-class communication.

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS) and the Graduate School "TrustSoft" (GRK 1076/1).

Train(*ID* : *TrainID*)

```
chan send, receive : [m : Message, id : TrainID]
chan computeSBI : [loa, sbi : Position]
local_chan selectSpeed, applyEB, applySB ···
```

The interface part of the class *Train* declares the channels *send* and *receive* for communication with the RBC. These channels carry two values indicating the kind of message and the ID of the sending train. In addition, we define the channel *computeSBI* carrying two position values. It computes the *service brake intervention limit* (SBI), i.e., the last position on the track the train has to apply the service brakes to halt before reaching the LOA. If the train exceeds this point the emergency brakes have to be used. The remaining channels (speed or brake instructions) are declared as *local*.

The external and internal **behavior** of CSP-OZ-DC components is described with CSP processes [2]. The processes communicate by sending data values over channels, respecting the channel declarations in the interface part. We call the occurrence of a communication an *event*. The process *Run* in the CSP part of *Train* determines a train's running behavior.

$$
\begin{aligned}
&\mathit{Run} \stackrel{c}{=} \mathit{updatePosition.ID?pos} \rightarrow \mathit{getLOA.ID?loa} \\
&\rightarrow \mathit{computeSBI!loa?sbi} \\
&\rightarrow \ \textbf{if } \mathit{sbi} \leq \mathit{pos} \\
&\quad \textbf{then } \mathit{applySB} \rightarrow \mathit{selectSpeed} \rightarrow \mathit{Run} \\
&\quad \textbf{else } \mathit{releaseSB} \rightarrow \mathit{selectSpeed} \rightarrow \mathit{Run}
\end{aligned}
$$

It states that every *updatePosition* event is followed by a *getLOA* event that updates the LOA, the position up to which the train may proceed without reaching the preceding train. The next permitted event, *computeSBI*, takes the new LOA as argument and computes the SBI. If the current position of the train is already beyond the SBI the train has to brake down. To this end, it selects a new speed value. The process *Run* is interleaved with a process *HandleEM* implementing the response of the train to emergency warnings.

Data aspects are specified with the object-oriented language Object-Z (OZ) [3]. The OZ part of every class consists of three kinds of schemata. The *state schema* defines the attributes of the class together with invariants restricting their values. The state schema of *Train* defines amongst others the current position and the SBI as attributes:

$\mathit{position} : \mathit{Position}$
$\mathit{sbi} : \mathit{Position}, \cdots$
$\mathit{position} < \mathit{sbi}$

com_*computeSBI*

$\Delta(\mathit{sbi})$ $\quad \mathit{loa}?, \mathit{sbi}! : \mathit{Position}$
$\mathit{sbi}! = \mathit{sbi}' =$
$\mathit{loa}? - \mathit{StopDist}$

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

An *initial schema* (not depicted here) constrains the attributes' initial values. *Operation schemata* associated with channels specify data changes that are performed at the moment the CSP part communicates over the channel. The operation schema for *computeSBI* contains a Δ expression indicating that the *sbi* attribute is changed by this schema. The input *loa*? is provided by the environment and contains the LOA position. The new value for *sbi* is represented by the primed variable *sbi*′, the return value by *sbi*!. These values are calculated depending on *loa*? and the distance *StopDist* the train needs to stop when exceeding the SBI.

We use an uninterpreted constant for the distance *StopDist*, i.e., this value is restricted by constraints, but not defined explicitly. Furthermore, our case study comprises variables of infinite data types like *position* and *speed*. We model the domains as reals: $\mathit{Position} == \mathbb{R}, \mathit{Speed} == \mathbb{R}_+$. The values of these variables are also transferred via channels.

Real-time constraints are described using the dense real-time logic DC [4]. Since we aim at automatic verification, we use the subclass of counterexample-trace formulae [1].

$$\neg(\text{true} \,;\, \updownarrow \mathit{receive.warning.ID} \,;\, 0.5 < \ell \wedge \boxminus \mathit{applyEB})$$
$$\ldots$$

The given DC formula states that after receiving a *warning* event ($\updownarrow$) we do *not* allow an interval greater than 0.5 time units ($0.5 < \ell$) to follow without ($\boxminus$) an *applyEB* event.

III. Verification of the Case Study

The formal semantics of CSP-OZ-DC is given in terms of timed automata extended with data variables, so-called phase event automata (PEA). Each part of a CSP-OZ-DC specification is translated into a single PEA [1]. A distinguished parallel composition is defined for PEA such that they synchronize on both events and data variables. This operation allows for a compositional semantics and it guarantees that once a safety property holds for one PEA it also holds for every parallel composition with this PEA.

In order to check a (safety) property given as DC formula for a CSP-OZ-DC model, we use the automata theoretic approach of Vardi and Wolper [8]. We first compute the PEA semantics of the model. The given DC formula, that has to be a counterexample-trace formula, is automatically transformed (the tool is available on [9]) into a set of test automata, i.e., PEA with distinguished bad states. The model satisfies the formula on an interval iff a bad state is reachable in the parallel composition of the PEA semantics and the test automata (cf. [10]). To cope with infinite data types and uninterpreted constants, we apply the abstraction refinement model checker ARMC [11] to check reachability in the product automaton.

Our aim is to ensure that two trains never collide. We express this property in the DC formula $\neg(\text{true} \,;\, \lceil \mathit{position}_1 > \mathit{position}_2 - \mathit{Length} \rceil)$. It shows that the property crucially depends on the trains' data behavior. The model of the full case study is too large to verify the property in a single step. However, we benefit from the compositionality of the CSP-OZ-DC semantics and verify local properties for the parallel components of our model. The compositional semantics ensures that those properties hold for the entire system.

TABLE I

Experimental results (Athlon XP 2200+, 512 MB RAM)

Task	Locs	Trans	Vars	Preds	Abstr	Refs	TA	ARMC
Run	178	6196	31	46	347	22	24.6	1556
Delivery 1	122	18757	122	472	2198	420	50	86874
Delivery 2	14	366	14	9	29	8	2.7	13.9
Delivery 3	17	173	10	9	27	5	2.2	1.9
Braking 1	44	240	17	6	61	5	3	5.1
Braking 2	172	1643	33	9	157	7	9	35.3

Table I gives a subset of our experimental results for a range of verification tasks. It lists the numbers of program locations (Locs), transitions (Trans), variables (Vars), predicates generated by ARMC (Preds), abstract states (Abstr), refinements loops (Refs), runtimes in seconds for generating test automata and parallel product (TA), and for model checking (ARMC). For instance, we consider the running behavior of the train in isolation and verify the safety property stated above, assuming that the first train does not apply the emergency brakes. To this end, we take only those PEA into account that influence the running behavior, i.e., the automata for the subprocess *Run* together with the appropriate automata for the OZ and the DC part. The result of this verification task is listed as "Run" in Table I. The table additionally lists the remaining verification tasks for the decomposition of our safety property.

From our case study, we experienced that decomposition of models as well as properties is necessary to handle industrial-scale systems. Therefore, the key prerequisite in our verification approach is the compositional semantics preserving the subcomponents' properties for the full system. Our results also demonstrate that the automata theoretic approach presented here is well-suited to check large-scale models (>18000 transitions) having infinite data types and uninterpreted constants. This is important for properties that cannot be decomposed further and depend on a number of components. The separate modeling of control flow, data part, and real-time requirements helped us handle the complexity of our case study.

References

[1] J. Hoenicke and P. Maier, "Model-checking of specifications integrating processes, data and time," in *FM 2005*, ser. LNCS, vol. 3582. Springer-Verlag, 2005, pp. 465–480.

[2] C. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1985.

[3] G. Smith, *The Object-Z Specification Language*. Kluwer, 2000.

[4] C. Zhou and M. R. Hansen, *Duration Calculus*. Springer-Verlag, 2004.

[5] ERTMS User Group, UNISIG, "ERTMS/ETCS System requirements specification," http://www.aeif.org/ccm/default.asp, 2002, version 2.2.2.

[6] H. Hermanns, D. N. Jansen, and Y. S. Usenko, "From StoCharts to MoDeST: a comparative reliability analysis of train radio communications," in *Workshop on Software and Performance*. ACM Press, 2005.

[7] A. Zimmermann and G. Hommel, "Towards modeling and evaluation of ETCS real-time communication and operation," *The Journal of Systems and Software*, vol. 77, no. 1, pp. 47–54, 2005.

[8] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Proceedings of the Symposium on Logic in Computer Science*. IEEE Computer Society, 1986, pp. 332–344.

[9] J. Hoenicke, R. Meyer, and J. Faber, "PEA toolkit home page," http://csd.informatik.uni-oldenburg.de/projects/epea.html, 2006.

[10] R. Meyer, J. Faber, and A. Rybalchenko, "Model checking duration calculus: a practical approach," in *ICTAC '06*, ser. LNCS, K. Barkaoui, A. Cavalcanti, and A. Cerone, Eds., 2006, to appear.

[11] A. Rybalchenko, "ARMC," http://www.mpi-inf.mpg.de/~ryb

Reducing Verification Complexity of a Multicore Coherence Protocol Using Assume/Guarantee

Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan
School of Computing, University of Utah
{xiachen, yuyang, ganesh}@cs.utah.edu

Ching-Tsun Chou
Intel Corporation
ching-tsun.chou@intel.com

***Abstract*— We illustrate how to employ metacircular assume/guarantee reasoning to reduce the verification complexity of finite instances of protocols for safety, using nothing more than an explicit state model checker. The formal underpinnings of our method are based on establishing a simulation relation between the given protocol M, and several overapproximations thereof, $\tilde{M}_1, \ldots, \tilde{M}_k$. Each $\tilde{M}_i$ simulates M, and represents one "view" of it. The $\tilde{M}_i$s depend on each other both to define the abstractions as well as to justify them. We show that in case of our hierarchical coherence protocol, its designer could easily construct each of the $\tilde{M}_i$ in a counterexample guided manner. This approach is practical, considerably reduces the verification complexity, and has been successfully applied to a complex hierarchical multicore cache coherence protocol which could not be verified through traditional model checking.**

I. Introduction

The dream of parameterized infinite-state verification of protocols for assertions written in expressive property languages such as CTL* are commonly held, and even demonstrated on actual protocols. However, highly complex industrial cache coherence protocols are the other extreme: they are so complex that one would be lucky to model check even small instances consisting of only a few caching agents for safety properties. With the imminence of multicore chips, the situation is expected to become worse, as multiple clusters of caching agents will interact through a second level of protocols, which will also be complex. The combined state space of protocols at various caching levels is astronomical. While one may attempt to separately verify each level and somehow "glue" together the results, discovering suitable formal arguments supporting the gluing step can, in fact, be an equally complex task. This paper addresses the following problems, making the indicated contributions:

• *There is no public domain hierarchical cache coherence protocol of reasonable complexity to employ as a verification benchmark.* In response to this problem, we developed a complex hierarchical protocol under the supervision of an industrial expert to ensure that our assumptions are realistic. Our protocol adapts the FLASH [1] protocol with MESI [2] features to manage caching within a cluster (consisting of two cache agents and the associated local directory), and adapts the DASH [3] protocol also with MESI features to manage coherence among three clusters. Figure 1 shows our verification model. It consists of a home cluster, h, and two identical remote clusters, r_1 and r_2, namely $M = (h \parallel r_1 \parallel r_2)$. Written in Murphi [4], the model [5] has about $3,000$ lines of code, and has the kind of complex scenarios that industrial protocols being designed today have.

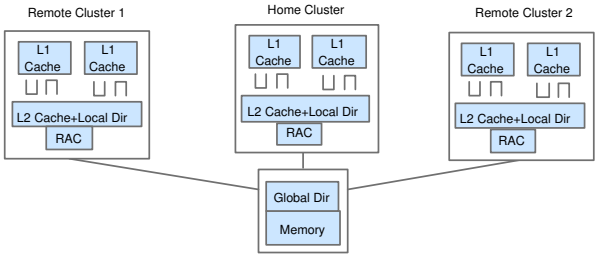


Fig. 1. A 2-level hierarchical cache coherence protocol.

• After removing many shallow bugs from M through non-exhaustive safety model checking, we failed to show the set of cache coherence properties ϕ_{coh} on M, due to state explosion, after $161,876,000$ of states. Our main contribution in this paper is a novel solution to this verification problem, adapting many ideas from a recently proposed assume/guarantee (A/G) approach. This A/G approach was first proposed in [6], but employed for *parameterized* verification in that work. Our method works as follows.

1. A set of abstracted protocols, $\tilde{M}_1, \ldots, \tilde{M}_k$, are constructed from M by simply projecting out (unconstraining) selected global variables, and correspondingly overapproximating the protocol rules.[1] Different variables are projected out for each M_i, and therefore, each $\tilde{M}_i$ presents a different overapproximated view of M.
2. Upon model checking each $\tilde{M}_i$ with respect to ϕ_{coh} and a set of non-interference lemmas (initially an empty set), there are three outcomes possible: *the verification succeeds*: M can then be claimed to satisfy ϕ_{coh}; *a genuine counterexample is generated:* the designer corrects M and iterates; *a counterexample infeasible in M is generated:* the designer strengthens some of the guards in $\tilde{M}_i$, and also adds a corresponding *verification obligation* (sometimes called *noninterference lemma*) to one of the $\tilde{M}_1, \ldots, \tilde{M}_k$, and the whole process is repeated.

Applying this procedure to our example protocol M, the above process generates three abstracted protocols, $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ ($\tilde{M}_2$ and $\tilde{M}_3$ are the same as explained in Section III). A genuine bug was found in M, and corrected. Thereafter,

[1] Murphi is a rule-based system, in which a transition relation is r
as: $rule$ "$name$" $guard \rightarrow action$;.

This work was supported in part by SRC Contract 2005-TJ-1318.

10 iterations were applied to both $\tilde{M}_1$ and $\tilde{M}_2$ to eliminate infeasible counterexamples. In the end, model checking was able to verify $\tilde{M}_1$ and $\tilde{M}_2$ with the same resources, over three and seven hours respectively[2].

- The refinement process discussed for $\tilde{M}_1$ and $\tilde{M}_2$ only requires modest manual effort, with each guard strengthening condition corresponding to a reasonably natural designer insight. Also, all noninterference lemmas can be obtained from the strengthening conditions, and hence the soundness of our method is assured (formally proved in Section IV).
- Our technique can be employed using *any* safety model checker.

The rest of the paper is organized as follows. Section II presents an overview and some features of the 2-level MESI protocol. Section III presents the generation of the $\tilde{M}_i$s, and Section IV describes the construction of the noninterference lemmas and guard-strengthening. Related work and concluding remarks follow.

II. Benchmark Hierarchical Coherence Protocols

Our benchmark hierarchical coherence protocol is derived by combining features from the FLASH and DASH protocols. Such a protocol is realistic, as DASH was originally developed for managing coherence across many clusters. It also renders our hierarchical protocol easy to understand for researchers who might want to attack this verification challenge problem. One address is modeled in our protocol, as is typical in model-checking based verification for coherence. As shown in Figure 1, the protocol is composed of three NUMA (Non-Uniform Memory Access) clusters: one home cluster and two identical remote clusters. Each cluster has two symmetric L1 caches, an L2 cache and a local directory. "RAC" is the communication controller with other clusters and the global directory. The main memory in reality is attached to every cluster. The fact there is only one memory is a consequence of the 1-address abstraction of our protocol. Finally, the global directory is to manage data copies on the three clusters.

In the 2-level hierarchy, the level-1 protocol is used within a cluster, i.e. the two L1 caches and the L2 cache in Figure 1. It tracks which line is cached in what state at which agent(s). The FLASH protocol is adapted to model this level and keep data copies within a cluster consistent. The level-2 protocol is used among clusters, tracking caching status in cluster level. The DASH protocol is adapted to keep clusters consistent. Also,

- For each cache line, if it is cached in an agent then it must also be cached in the local directory of the agent, i.e. the *inclusive* property;
- Both levels use MESI, supporting explicit write-back and silent-drop[3]
- Both levels use non-FIFO network ordering (this, according to our industrial expert, is a desirable feature of modern protocols).

In the process of developing our hierarchical protocol, we discovered that it was not simply a matter of adapting FLASH and DASH to support MESI states and silent-drops, as such combinations can easily lead to livelocks. One such scenario is the following. Agent-1 of cluster-1 first requests an exclusive copy and gets granted. As a result, a subsequent request from the rest of the system to the same line will be forwarded to agent-1. At this time, if agent-1 has silently dropped the cache line, it will NACK the forwarded request. Because the local directory of agent-1 has no information about the silent-drop happening in agent-1, and it is not safe to use the data copy in the local directory to reply (e.g. write-back from agent-1 is on the way), cluster-1 will keep forwarding following requests to agent-1 and agent-1 will keep NACKing. This results in a livelock. We solved this livelock problem by making write-back a blocking operation and adding another NACKing message indicating silent-drops, as detailed in [5].

```
ProcState: record
   -- B1 agents in the cluster
   Proc :   array [NODE] of NODE_STATE;

   -- B2 network channels used in the cluster
   UniMsg:  array [NODE_L2] of UNI_MSG;
   InvMsg:  array [NODE_L2] of INV_MSG;
   WbMsg:   WB_MSG;
   ShWbMsg: SHWB_MSG;
   NakcMsg: NAKC_MSG;

   -- local dir for the cluster
   L2: record
       -- B3.1 used only by level-1 protocol
       pending:    boolean;
       ShrSet:      array [NODE] of boolean ;
       InvCnt:     CacheCnt;
       HeadPtr:    NODE_L2;
       ReqId:      NODE;
       ReqCluster: Procss;
       ReqType:    boolean;
       isRetired:  boolean;
       ifHoldMsg:  boolean;
       -- B3.2 used by both levels
       State:     L2State;
       Data:      Datas;
       Dirty:     boolean;
       OnlyCopy:  boolean;
       Gblock_WB: boolean;
   end;

   -- B4 comm. controller with other clusters and global dir
   RAC: record
       State:    RACState;
       InvCnt:  ClusterCnt;
   end;
end;
```

```
ABSProcState: record
    L2: record
         -- B3.2
        State:     L2State;
        Data:      Datas;
        Dirty:     boolean;
        OnlyCopy:  boolean;
        Gblock_WB: boolean;
    end;

     -- B4
    RAC: record
        State:    RACState;
        InvCnt:  ClusterCnt;
    end;
end;
```

Fig. 2. The concrete and abstract structures in Murphi, representing a cluster in the hierarchical protocol.

[2]These large run-times are due to the relatively complex nature of the protocol compared to many traditional academic benchmarks. Also, no hash compaction was used, as the in-house 64-bit version of Murphi we employed has not been tested under hash compaction.

[3]Silent-drop in MESI protocols means a cache may discard a non-*M*odified line at any time, changing to the *I*nvalid state without informing the local or global directory.

Our hierarchical coherence protocol coded in Murphi includes all the control logic and data coherence invariants that FLASH and DASH have. In Figure 2, we present the data structure representing a cluster in the hierarchical protocol, the record "ProcState". It contains five blocks of information: (**B1**) number "NODE" of agents in a cluster, (**B2**) a set of network channels used inside a cluster, (**B3.1**) & (**B3.2**) the local directory, and (**B4**) a controller to communicate with other clusters and the global directory. In these five blocks, the level-1 protocol (used within a cluster) only involves the first four blocks, i.e. blocks $B1$, $B2$, $B3.1$ and $B3.2$, and the level-2 protocol (used among clusters) only involves blocks $B3.2$ and $B4$. We also present the data structure of an abstracted cluster in Figure 2, the record "ABSProcsState", as it will be used quite often in the rest of the paper. An abstracted cluster only contains blocks $B3.2$ and $B4$.

III. Building abstracted protocols

Our hierarchical protocol proved to be very complex such that after $161,876,000$ states, the model checking failed due to state explosion. This is not surprising, considering the multiplicative effect of having three instances of coherence protocols running concurrently. We believe that all hierarchical (e.g., multicore) coherence protocols will state explode in this manner. This section describes how our initial abstracted models $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ were obtained. Due to the symmetry between the two remote clusters, $\tilde{M}_2$ is actually the same with $\tilde{M}_3$. So only $\tilde{M}_1$ and $\tilde{M}_2$ will be discussed in the rest of the paper.

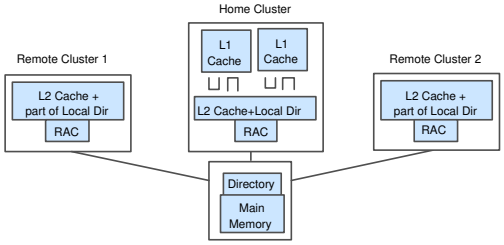


Fig. 3. The abstracted protocol $\tilde{M}_1$

A. Building the abstract models

Figure 3 intuitively depicts the abstracted protocol $\tilde{M}_1$, which retains the home cluster intact, while abstracting the two remote clusters (contrast with Figure 1). The abstracted protocol $\tilde{M}_2$ is very similar to $\tilde{M}_1$, except that it retains one remote cluster concretely and abstracts the other remote cluster as well as the home cluster. The derivation of $\tilde{M}_1$ and $\tilde{M}_2$ from M involves the variable abstraction and the corresponding transition relation and invariant abstraction described next. Specifically, in M, "Home" is the singleton set containing the home cluster and "Rmt" is the set containing the remote clusters [4]. "Procss" is their union, and the clusters are declared as "Procs:"

```
HomeCnt: 1;
Home:    scalarset(HomeCnt);
RmtCnt:  2;
Rmt:     scalarset(RmtCnt);
Procss:  union {Home, Rmt};

Procs: array [Procss] of ProcState
```

[4]We employ scalarsets to obtain the benefit of symmetry reduction.

Now, let us consider the state declarations of clusters in $\tilde{M}_1$ (first half of Figure 4) and $\tilde{M}_2$ (second half of Figure 4). In $\tilde{M}_1$, the home cluster will still be declared as "ProcState," while the remote clusters will be declared as "ABSProcState," the data structure only containing part of the local directory and the communication controller of a cluster. $\tilde{M}_2$ is similar.

```
-- clusters in M1
HomeCnt: 1;
RmtCnt:  2;
Home:    scalarset(HomeCnt);
Rmt:     scalarset(RmtCnt);
Procss:  union {Home, Rmt};

Procs:    array [Home] of ProcState;
ABSProcs: array [Rmt] of ABSProcState;
```

```
-- clusters in M2
HomeCnt:  1;
RmtCnt_1: 1;
RmtCnt_2: 1;
Home:     scalarset(HomeCnt);
Rmt_1:    scalarset(RmtCnt_1);
Rmt_2:    scalarset(RmtCnt_2);

Procs:   array [Rmt_1] of ProcState;
ABSHome: array [Home] of ABSProcState;
ABSRmt:  array [Rmt_2] of ABSProcState;
```

Fig. 4. Declaration of clusters in $\tilde{M}_1$ and $\tilde{M}_2$

For each transition (*TR*) in M, a *set* of corresponding TRs are constructed in $\tilde{M}_1$ and $\tilde{M}_2$. These could be automatically generated, and we have described the procedure for $\tilde{M}_1$ in the appendix (the procedure for $\tilde{M}_2$ is similar). We consider the rules one at a time in M. For every rule "$guard \rightarrow action$", for any assignment of the form `v := E` in $action$, (i) if `v` is a variable that has been eliminated (abstracted away), then the whole assignment is eliminated, (ii) else if `E` contains even one variable that has been abstracted away, we replace `E` with a non-deterministic selection over the type of `E`. If a sub-expression in $guard$ contains any variable that has been abstracted away, the sub-expression turns into $true$.

IV. Verifying the hierarchical protocol

This section presents details of the refinement process and the soundness of our approach.

A. Counterexamples, lemmas, and guard-strengthening

Figure 5 shows the process of how the refinement is applied on $\tilde{M}_1$, $\tilde{M}_2$ and M (if it is buggy): When $\tilde{M}_1$ is model checked using Murphi with respect to ϕ_{coh}, if a genuine bug is detected, the designer corrects and reiterates. Assume that the error is a false alarm, and involves $rule_p$ "$g_p \rightarrow a_p$" of $\tilde{M}_1$. The corresponding rule in M is then located; assume it is "$G_p \rightarrow A_p$". Then, based on the expression G_p, we *manually* derive another expression i_p which only contains the variab

in the level-2 protocol, i.e. variables in blocks $B3.2$ and $B4$ in a cluster, the global directory, and the set of network channels used among clusters. We will discuss how such an expression i_p can be derived from G_p in a detailed example in the next section.

Now we add a *new verification obligation* (noninterference lemma) "$G_p \Rightarrow i_p$" to $\tilde{M}_2$, and at the same time strengthen $rule_p$ in $\tilde{M}_1$ to be "$g_p \wedge i_p \rightarrow a_p$".

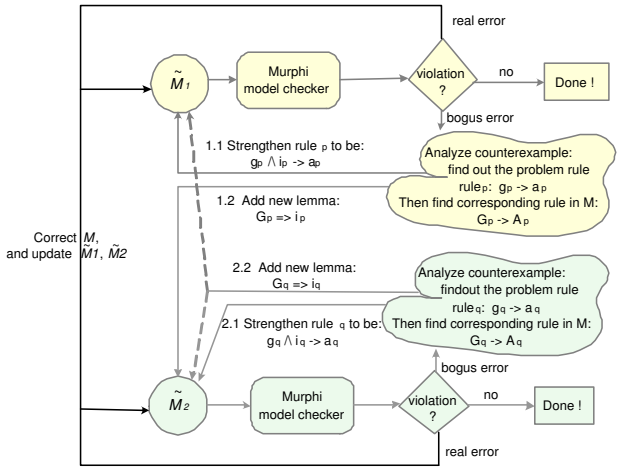


Fig. 5. Counterexample-guided metacircular refinement.

It is perhaps surprising that the noninterference lemma is added as a verification obligation in $\tilde{M}_2$, while the consequent of this lemma, i_p, is used in $\tilde{M}_1$. This is because according to the abstraction procedure in Section III, we know that the bogus error in $\tilde{M}_1$ is introduced by the overapproximation on the remote clusters. In particular, G_p contains some remote cluster details, however they are abstracted away in g_p. So the noninterference lemma $G_p \Rightarrow i_p$ can only be checked in $\tilde{M}_2$ where all the state elements present in G_p are available. A similar process can be applied while refining $\tilde{M}_2$, as also shown in Figure 5. Essentially, for each noninterference lemma L, it suffices to prove L in any of the abstracted models. The question of in which abstracted model to prove L is determined by which abstracted model has enough details for L to be proved.

B. A detailed example of refinement

One counterexample encountered in $\tilde{M}_1$ is as follows: after the startstate rule is fired (e.g. the state is just initialized), Rule "L2_Recv_NAKC_Nakc" in $\tilde{M}_1$ is fired and the first statement in the action – the assertion – is violated. This rule is shown in the second half of Figure 6, and the corresponding rule in M is also shown in the first half of the figure.

Consider the condition under which this rule can be enabled in the hierarchical protocol M. Figure 7 shows an example scenario: The remote cluster "p" initially holds an exclusive copy in the agent "src," another remote cluster "aux" requesting for an exclusive copy (step 1 and 2) is forwarded by the global directory to the remote cluster "p" (step 3), and because the local directory of "p" indicates that the agent "src" holds the exclusive copy, "p" will forward the request to "src" (step 4.1); concurrently, "src" silently drops the copy (step 4.2). So in receiving the forwarded request from the local directory, "src" NACKs the request (step 5).

```
ruleset p: Rmt; src: NODE do
rule "L2_Recv_NAKC_Nakc"
    Procs[p].NakcMsg.Cmd = NAKC_Nakc &
    Procs[p].NakcMsg.Aux = L2 &
    Procs[p].L2.ReqType = false
==>
var aux: Rmt;
begin
    assert (Procs[p].L2.pending = true);
    Procs[p].L2.pending := false;

    assert(!isundefined(Procs[p].L2.ReqCluster));
    aux := Procs[p].L2.ReqCluster;

    assert (Procs[p].L2.ifHoldMsg = true);
    Procs[p].L2.ifHoldMsg := false;
    assert (GUniMsg[aux].Cmd = RDX_RAC &
            GUniMsg[aux].Cluster = p);
    undefine GUniMsg[aux];
    GUniMsg[aux].Cmd := NAK;

    GNakcMsg.Cmd := GNAKC_Nakc;
    GNakcMsg.Cluster := p;
    undefine Procs[p].L2.ReqCluster;
    undefine Procs[p].L2.ReqType;

    undefine Procs[p].NakcMsg;
    Procs[p].NakcMsg.Cmd := NAKC_None;
endrule;
endruleset;
```

Rule "L2_Recv_NAKC_Nakc" in M

```
ruleset p: Rmt; aux: Rmt do
rule "L2_Recv_NAKC_Nakc"
    true
==>
begin
    assert (GUniMsg[aux].Cmd = RDX_RAC&
            GUniMsg[aux].Cluster = p);
    undefine GUniMsg[aux];
    GUniMsg[aux].Cmd := NAK;

    GNakcMsg.Cmd := GNAKC_Nakc;
    GNakcMsg.Cluster := p;
endrule;
endruleset;
```

Rule "L2_Recv_NAKC_Nakc" in $\tilde{M}_1$

Fig. 6. Rule "L2_Recv_NAKC_Nakc" in M and $\tilde{M}_1$.

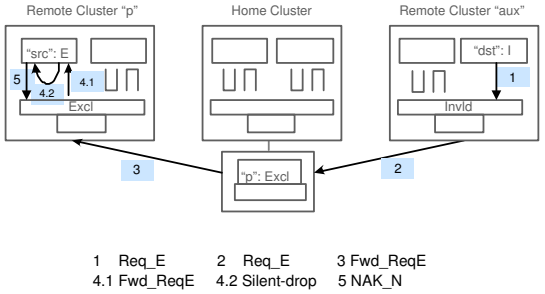


Fig. 7. A scenario which can enable "L2_Recv_NAKC_Nakc" in M.

Obviously, we can see that the violation in $\tilde{M}_1$ is a bogus error. The reason is because the guard condition of Rule "L2_Recv_NAKC_Nakc" is abstracted to "$true$", which is overly approximated compared with that in M. Figure 8 shows the solution we used to overcome this violation: a new lemma "Lemma-7-A2" is added to $\tilde{M}_2$, and the consequent of this lemma is used to strengthen the g

"L2_Recv_NAKC_Nakc" in $\tilde{M}_1$.

```
ruleset p: Rmt; aux: Procss do
rule "L2_Recv_NAKC_Nakc"
    true &
    --  Lemma-7-A2
    ABSProcs[p].L2.State = Excl &
    ABSProcs[p].RAC.State = Inval &
    ABSProcs[p].L2.Gblock_WB = false &
    GUniMsg[aux].Cmd = RDX_RAC &
    GUniMsg[aux].Cluster = p
==>
begin
    ...
end;
```

Guard strengthening in $\tilde{M}_1$ using lemma-7-A2

```
invariant "Lemma-7-A2"
forall p: Rmt do
    Procs[p].NakcMsg.Cmd = NAKC_Nakc &
    Procs[p].NakcMsg.Aux = L2 &
    Procs[p].L2.ReqType = false
->
    Procs[p].L2.State = Excl &
    Procs[p].RAC.State = Inval &
    Procs[p].L2.Gblock_WB = false &
    exists aux: Procss do
      GUniMsg[aux].Cmd = RDX_RAC &
      GUniMsg[aux].Cluster = p  end
end;
```

New invariant added to $\tilde{M}_2$

Fig. 8. New "Lemma-7-A2" of $\tilde{M}_2$ and guard-strengthening in $\tilde{M}_1$.

Intuitively, the newly added lemma tries to mimic the level-2 protocol state from the available level-1 protocol state. In the example of Figure 8, "Lemma-7-A2" says when the local directory of a remote cluster receives "NAKC_Nakc" and this negative reply is for a forwarded request from another remote cluster, i.e. "Procs[p].NakcMsg.Aux = L2", then part of the level-2 state must be as follows – the remote cluster "p" is in Exclusive state, its communication controller is free, it is not blocked on write-back, and there exists another remote cluster "aux" requesting an exclusive copy.

C. Refinement results

In the refinement process, we found a real bug in M. The scenario corresponding to the real bug in M is similar to that in Figure 7. Initially, a cluster called p holds an exclusive copy in the agent src; $(S1)$ p receives a forwarded request from another cluster aux, for an exclusive copy; $(S2)$ p then forwards the request to src, the owner agent of p. $(S3)$ Concurrently, src updates the cache line and $(S4)$ writes back the dirty line to p; $(S5)$ therefore, upon receiving the forwarded request, src will NACK it. $(S6)$ When p receives the write-back data and then $(S7)$ receives the NACKed reply from src, p will $(S8.1)$ reply positively to aux with a data, also $(S8.2)$ send a message to the global directory notifying that aux is now the exclusive owner.

In $(S8.1)$ of the above scenario, the cluster p should indicate whether the data supplied to aux is dirty or not, because this data is not sent to the global directory in $(S8.2)$. If there is no such indication, then aux, when it receives the data, can set the state of the data to Exclusive. This permits aux to later silently drop the cache line. This, in effect, throws away the only copy of the (most recently modified) data, violating the coherence protocol.

We eliminated this bug in M by attaching a dirty tag to the reply message in $(S8.1)$. Correspondingly, $\tilde{M}_1$ and $\tilde{M}_2$ are updated for this modification. After that, we also added 10 new noninterference lemmas to $\tilde{M}_1$ and $\tilde{M}_2$ individually.

The following describes the 10 new noninterference lemmas[5] added to $\tilde{M}_1$ and $\tilde{M}_2$ after the above bug was corrected.
Step 1: In $\tilde{M}_1$, a short counterexample is encountered, which violates the assertion that when a cluster receives a write-back from an agent, the cluster must be in Exclusive state. This violation is introduced in $\tilde{M}_1$ because the guard condition of receiving write-back in remote clusters is abstracted to "$true$". Correspondingly, two similar counterexamples are encountered in $\tilde{M}_2$, with the same violation on the abstracted home cluster and the abstracted remote cluster. One lemma is added to $\tilde{M}_2$, and its consequent is used in $\tilde{M}_1$ and $\tilde{M}_2$. Another lemma is added to $\tilde{M}_1$, but used in $\tilde{M}_2$, both asserting that when a cluster receives a write-back request, the cluster must be Exclusive.
Steps 2,3,6,7: Similar to step 7, which is already discussed in Section IV-B.
Step 4: Three short counterexamples are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating an invariant that when a cluster is dirty or Exclusive, other clusters must be Invalid for the same cache line. The fix asserts that when a cluster receives a shared write-back request from an agent, then that agent must be Exclusive.
Step 5: Three counterexample are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating the assertion of an impossible branch in the hierarchical protocol M ("assert false"). The fix involves asserting that: if there is a state s in which (i) a cluster receives a NACK reply indicating the silent dropping of a line by an agent, *and* (ii) when such NACK arrives, the exclusive owner pointer is pointing neither at the local directory nor at the agent sending the NACK, then s is impossible.
Step 8: Three rather long counterexamples are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating the invariant that when a cluster is waiting for invalidation acknowledgments and another cluster is Shared for the line, then there must exist an invalidation request being sent to the second cluster. A similar fix is applied, asserting that when the owner pointer of a cluster is the local directory, the cluster must be either Exclusive or Shared.
Step 9: Three counterexamples are encountered in $\tilde{M}_1$ and $\tilde{M}_2$, violating the invariant that when the global directory is not Exclusive, the main memory data should be the same with the current value of the cache line in the system. A similar fix is applied, asserting that when a cluster receives a shared request from a second cluster, the first cluster must be Exclusive.
Step 10: Three counterexamples are encountered, violating the assertion that when an exclusive request is granted, the reply should also contain the data. A similar fix as in Step 9 is applied.

After Step 10, all counterexamples disappear. Model checking on $\tilde{M}_1$ results in $31,919,219$ states using three hours, and

[5]These 10 refinement steps are listed only to give the reader a detailed glimpse at the nature of the errors and their strengthening condi[illegible] The details are not important to follow.

$78,689,678$ states for $\tilde{M}_2$ using seven hours, both with the same resources.

D. A theorem justifying metacircular reasoning

The intuitive reason for the soundness of this approach is the following inductive argument. The verification obligations such as $G_p \Rightarrow i_p$ that are added to each of the abstract models $\tilde{M}_i$ are, of course, checked at the initial state. This implies that when G_p is true (the corresponding rule is enabled), i_p is implied. In other words, we *never* be underapproximating the state space if i_p is used to strengthen G_p. Since the valuations of the variables in $\tilde{M}_1$ and $\tilde{M}_2$ are the same as that in M, strengthening an *abstracted version* of G_p with i_p (which is what we do in our refinement method) is also sound. Now, since this check is done at every step, we ensure that the entire reachable state space is an overapproximation.

More formally, in [6], a theory was developed based on the classical notion of simulation proofs [7]. This theory was used to justify metacircular reasoning in *parameterized* verification of cache coherence protocols. However, the theory itself does not depend on any parameterized verification features. We summarize the main theorem of [6] as follows (we retain the equation and theorem numbers used in that paper):

Theorem 3. Suppose M is a state transition system (STS): $M = (S, I, T)$, where S is the set of states, $I \subseteq S$ is the set of initial states, $T \subseteq S \times S$ is the set of transition relations, and $\mathcal{R}(M)$ is the set of reachable states of M. Also suppose V is a set of "views" on M, $\{f_v : v \in V\}$ is a set of functions that create each view abstraction over the states in M, and $\{\tilde{M}_v : v \in V\}$ is the corresponding set of abstracted STS. Let f (respectively f^{-1}) be obtained by lifting f_v (respectively f_v^{-1}) over tuples of abstract states. If there exists an abstract system which is a product STS, $\tilde{M} = \Pi_{v \in V} \tilde{M}_v$, where $\tilde{M}_v = (\tilde{S}_v, \tilde{I}_v, \tilde{T}_v)$ for $v \in V$, then if for each $v \in V$:

(7) $\forall s \in I : f_v(s) \in \tilde{I}_v$

(8) $\forall (s, s') \in T : (\forall u \in V : f_u \in \mathcal{R}(\tilde{M}_v)) \Rightarrow f_v(s, s') \in \tilde{M}_v$

then $(f^{-1}(\mathcal{R}(\tilde{M})), f)$ is a simulation from M to $\tilde{M}$ and:

(9) $\forall s \in \mathcal{R}(M) : (\forall v \in V : f_v(s) \in \mathcal{R}(\tilde{M}_v))$

While space does not permit a detailed explanation, we can actually explain these results quite well using our hierarchical protocol M. It is easy to see that each state s in M can be represented as a state vector

$$s \;=\; [\langle h.1, h.2\rangle,\ \langle r1.1, r1.2\rangle,\ \langle r2.1, r2.2\rangle,\ \langle gs\rangle]$$

where $\langle h.1, h.2\rangle$, $\langle r1.1, r1.2\rangle$ and $\langle r2.1, r2.2\rangle$ are the state components over the home cluster, remote-1 and remote-2 clusters. Moreover, $h.1$, $r1.1$, $r2.1$ correspond to the blocks $B1$, $B2$ and $B3.1$ in $ProcState$ of Figure 2, and $h.2$, $r1.2$, $r2.2$ corresponds to the blocks $B3.2$ and $B4$ in $ProcState$. gs is the rest of information in s, including the global directory and the network channels used among clusters. Now we discuss *three* abstractions in this section, $\tilde{M}_1, \tilde{M}_2$ and $\tilde{M}_3$, as was the case in our discussions in Section III. We can have the following theorem and equations in our framework. All numberings now carry a prime.

Theorem 3'. If f_1, f_2, f_3 are three abstract functions s.t.

(7')

$\forall s \in I$ and
$s = [\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

$\Rightarrow$

$f_i(s) \in \tilde{I}_i,\ i \in [1..3]$, where
$f_1(s) = [\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_2(s) = [\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_3(s) = [\langle h.2\rangle, \langle r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

(8')

$\forall (s, s') \in T : f_i(s) \in \mathcal{R}(\tilde{M}_i),\ i \in [1..3]$ and
$s = [\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$
$s' = [\langle h.1', h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.1', r2.2'\rangle, \langle gs'\rangle]$

$\Rightarrow$

$f_i(s, s') \in \tilde{T}_i,\ i \in [1..3]$, where
$f_1(s') = [\langle h.1', h.2'\rangle, \langle r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
$f_2(s') = [\langle h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle]$
$f_3(s') = [\langle h.2'\rangle, \langle r1.2'\rangle, \langle r2.1', r2.2'\rangle, \langle gs'\rangle]$

then all the invariants in $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ are valid in M, including the coherence properties. □

The fact that *Theorem 3'* is indeed a theorem follows from the fact that the antecedents of *Theorem 3'* are stronger than the antecedents used in *Theorem 3*.

1) Applying the theorem: Now we prove that the abstracted protocols $\tilde{M}_1, \tilde{M}_2$ and $\tilde{M}_3$ ($\tilde{M}_3$ is the same with $\tilde{M}_2$ for our hierarchical protocol M) indeed satisfy the conditions (7') and (8'). This will then allow us to conclude that the cache coherence properties in the abstracted protocols also hold in the original hierarchical protocol. We will, therefore, focus only on (8'), as the proof of satisfaction of (7') is much simpler.

Proof:

From a simple induction, we have

(9)'

$\forall s \in \mathcal{R}(M)$,
$s = [\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

$\Rightarrow$

$f_1(s) = [\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_2(s) = [\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle]$
$f_3(s) = [\langle h.2\rangle, \langle r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle]$

Now, consider any state transition (s, s') in the hierarchical protocol M. Assume that s' is obtained by firing a rule in M, "$R : g \rightarrow a$". From Section III-A, we know there exists one rule corresponding to R in each of $\tilde{M}_1$, $\tilde{M}_2$, and $\tilde{M}_3$. These are: in $\tilde{M}_1$ "$R1 :\ g1 \rightarrow a1''$; in $\tilde{M}_2$ $''R2 :\ g2 \rightarrow a2''$, and in $\tilde{M}_3$ $''R3 :\ g3 \rightarrow a3''$.

Also, the guard expression g of R can belong to only one of the following four cases. This is because any level-1 protocol details can only be visible to the level-2 protocol in the *same* cluster:

1. g only involves variables in $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
2. g only involves $\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
3. g only involves $\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
4. g only involves $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle$

We now analyze each case to see that (8') is indeed satisfied.

Case 1. g only involves $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ (i.e. does not refer to any "inner" state components).

From Section III-A, we know that $g1 = g2 = g3 = g$, also because

$f_1(s), f_2(s), f_3(s)$ and s hold the same values over variables in $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$

So $f_1(s), f_2(s), f_3(s)$ are enabled at $R1, R2, R3$ individually; from Section III-A we also know that the action ai's in rule Ri's, have the exact updates over variables in $f_i(s),\ i \in [1..3]$, as the action of the hierarchical protocol, a in R does. So

- $a1(f_1(s)) = [\langle h.1', h.2'\rangle, \langle r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle] = f_1(s')$
- $a2(f_2(s)) = [\langle h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle] = f_2(s')$
- $a3(f_3(s)) = [\langle h.2'\rangle, \langle r1.2'\rangle, \langle r2.1', r2.2'\rangle, \langle gs'\rangle] = f_3(s')$

Case 2. g only involves $\langle h.1, h.2\rangle$,$\langle r1.2\rangle, \langle r2.2\rangle$,$\langle gs\rangle$ (i.e. g refers to some "inner" state components of the home cluster)

On one side, for $R1$ in $\tilde{M}_1$, based on Section III-A,

- $g1 = g$
- $a1$, the action of $R1$ in $\tilde{M}_1$, has the exact updates over the variables in $\langle h.1, h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ as a, the action of R in M does;

So $f_1(s)$ is enabled at $R1$, and

- $a1(f_1(s)) = [\langle h.1', h.2'\rangle, \langle r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle] = f_1(s')$

On the other side, for $R2$ in $\tilde{M}_2$, based on Section III-A,

$g2$ only involves variables in $\langle h.2\rangle$,$\langle r1.2\rangle, \langle r2.2\rangle$,$\langle gs\rangle$, and all the sub-expressions in $g2$ involving any of these variables are exactly the same as the sub-expressions in g, so $g \Rightarrow g2$; or a new lemma is added and guard-strengthening is applied such that

- $g2 = g2_0 \wedge G_{uard}S_{strength}$
- We do know that $g \Rightarrow g2_0$
- $G_{uard}S_{strength}$ only involves variables in $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$
- $g \Rightarrow G_{uard}S_{strength}$ (this follows from the non-interference lemma being established)

Please note that the above reasoning does not depend on whether the new lemma was added into $\tilde{M}_1$, $\tilde{M}_2$ or $\tilde{M}_3$. The only property that matters is that for each variable in $\langle h.1, h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.1, r2.2\rangle, \langle gs\rangle$, it has the same value (if exists) in $s, f_1(s), f_2(s)$ and $f_3(s)$, which has already been stated in (9').

Again, because the variables in $f_2(s)$ have the same values over $\langle h.2\rangle, \langle r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ with s, so

- $f_2(s) \Rightarrow g2_0$
- $f_2(s) \Rightarrow G_{uard}S_{strength}$

That is, $f_2(s)$ is enabled at $R2$. Based on Section III-A, we know that the action $a2$ in rule $R2$ of $\tilde{M}_2$, has the exact updates over state components $\langle h.2\rangle, \langle r1.1, r1.2\rangle, \langle r2.2\rangle, \langle gs\rangle$ as a, the action of R in M does. So

- $a2(f_2(s)) = [\langle h.2'\rangle, \langle r1.1', r1.2'\rangle, \langle r2.2'\rangle, \langle gs'\rangle] = f_2(s')$

Similarly, the result holds for f_3: $a3(f_3(s)) = f_3(s')$.

Case 3& 4. These two cases can be proved similarly. □

The above proof justifies that the coherence properties in the abstracted protocols $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$ also hold in the hierarchical protocol M. Moreover, according to Section III-A, we know that every invariant in M is covered by invariants in $\tilde{M}_1$ and $\tilde{M}_2$. So, it follows that once $\tilde{M}_1$ and $\tilde{M}_2$ are model checked to be coherent, the hierarchical protocol is also coherent.[6]

V. Related Work

This paper owes most of its intellectual debts to the work in [6] on verifying parameterized cache coherence protocols, and McMillan's work on compositional model checking [8]. The abstractions we used, the reliance on circular reasoning, and the counterexample-guided discovery of noninterference lemmas are all deeply influenced by their work. In addition to all our contributions pointed out earlier, we have two methodological contributions: Firstly, we find a way to naturally abstract a 2-level hierarchical cache coherence protocol into a few far simpler protocols. Secondly, the refinement process on the abstracted protocols is straightforward to conduct, including the noninterference lemmas. Finally, based on a theorem in [6], we have formally verified that our abstraction method is sound and complete.

McMillan [9] also modeled a 2-level MSI coherence protocol, based on the Gigamax distributed multiprocessor. In his protocol, bus-snooping is used in both levels. SMV [9] was used to check two clusters each having six processors with safety and liveness properties. Compared to our hierarchical protocol, the Gigamax model is much simpler.

Lahiri and Bryant in [10] use predicate abstraction to automatically construct quantified invariants. Their abstraction is similar with ours: for each concrete state, the abstraction maps it into multiple abstract states each of which corresponds to subranges of a set of universally quantified variables. Overapproximation ensures that the soundness of properties in the abstracted system guarantee the soundness in the concrete system.

VI. Conclusions and Future Work

Hierarchical cache coherence protocol verification is a challenging problem, as these protocols have many more protocol corner cases than non-hierarchical protocols, and have too many reachable states. In this paper, we propose a method to abstract a 2-level hierarchical coherence protocol into a few far simpler and tractable protocols. By verifying these simpler protocols, the correctness of the hierarchical protocol follows through the refinement theorem, also presented in this paper. Our success with the complex 2-level MESI hierarchical protocol developed in consultation with an industrial coherence protocol designer leads us to believe that other hierarchical coherence protocols – including snoopy, directory-based, ring interconnect based, as well as token-coherence based protocols – can be similarly verified.z Protocols with more than two

[6]The process of refinement will, of course, terminate because in the extreme case, we may end up describing the original protocol M [illegible] the noninterference lemmas! This extreme is not expected in pract

levels (again common in large shared memory machines) also appear entirely amenable to our approach.

We plan to mechanize our method as much as possible. After the designer picks the variables to be projected out, constructing the initial abstract protocols could be automated. Identifying the noninterference lemmas will, in general, force the designer to understand their protocol - albeit in a localized manner in response to the counterexamples. It would also be interesting to exploit the fact that the two abstract protocols have a common subset of transitions, e.g. the level-1 protocol which only involves the agents and the local directory within a cluster. Extending our approach for non-inclusive cache coherence protocols is also in progress.

Acknowledgments We would thank Ritwik Bhattacharya, Igor Melatti and Liqun Cheng for their help on this work.

REFERENCES

[1] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. G. J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, "The stanford flash multiprocessor," in *ISCA*, 1994.

[2] M. Papamarcos and J. Patel, "A low overhead coherence solution for multiprocessors with private cache memories," in *ISCA*, 1984.

[3] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *ISCA*, 1990.

[4] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[5] Http://www.cs.utah.edu/formal_verification/fmcad06models.tar.gz.

[6] C.-T. Chou, P. K. Mannava, and S. Park, "A simple method for parameterized verification of cache coherence protocols," in *FMCAD*, 2004.

[7] R. Milner, "An algebraic definition of simulation between programs," in *IJCAI*, 1971.

[8] K. McMillan, "Verification of infinite state systems by compositional model checking," in *CHARME*, 1999.

[9] K. L. McMillan, "Symbolic model checking," Ph.D. dissertation, Carnegie Mellon University, 1992.

[10] S. K. Lahiri and R. E. Bryant, "Constructing quantified invariants via predicate abstraction," in *VMCAI*, 2004.

APPENDIX
PROCEDURE FOR ABSTRACTING THE TRANSITION RELATION

We only present the procedure to construct the transition relations for $\tilde{M}_1$ in the following, as the procedure for $\tilde{M}_2$ is very similar. Our procedure is described in the context of Murphi, but the ideas are broadly applicable. We denote the set of variables which are abstracted away in $\tilde{M}_1$ by D.

1) Pre-processing:
   a) If there exists a $ruleset$ parameter whose range is over all the clusters in M, i.e. "Procss", divide the rule into two rules with the same guard and action: one rule with the parameter only over the home cluster ("Home"), and the other with the parameter over the remote clusters ("Rmt").
   b) If there exists any conditional statements, i.e. ifstmt or switchstmt, in the action of a rule, divide the rule into several sub-rules such that each sub-rule covers one branch of the conditional statement and the sub-rule does not contain conditional statements anymore. The guard of the sub-rule is the logic "$\wedge$" of the original rule guard and the conditional expr(s) for that branch. The action is composed by the statements inside the branch and the rest of statements in the original rule action.
2) For each rule with $ruleset$ parameters only over the home cluster, do nothing for the rule.
3) For each rule with $ruleset$ parameters over the remote clusters, or without parameters
   a) If there exists any $ruleset$ parameters over D, remove such parameters.
   b) For the rule guard, replace "Procs[]" over remote clusters with "ABSProcs[]", i.e. replacing concrete clusters with abstract clusters; if there exists any boolean sub-expr involving "$forall$" or "$exists$" with parameters over all the clusters, replace the sub-expr with two "$\wedge$" sub-exprs: one with the parameter over the home cluster, and the other with the parameter over the remote clusters; if there exists any boolean sub-expr involving variables in D, replace the sub-expr with "$true$".
   c) For each statement in the rule action, replace "Procs[]" over remote clusters with "ABSProcs[]",
      i) assignment: if the designator involves any variable in D, remove the assignment; otherwise, for the expr to be assigned to the designator, if there exists any sub-expr involving variables in D, replace the sub-expr with a nondeterministic value in the type of the sub-expr.
      ii) forstmt: if the quantifier in the forstmt ranges over all the clusters, divide the forstmt into two forstmts: one with the quantifier on the home cluster, and the other on the remote clusters; if the quantifier involves any variable in D, remove the forstmt; for each statement inside the forstmt, goto 3(c).
      iii) whilestmt: for the condition expr in the whilestmt, if there exists any sub-expr involving variables in D, replace the sub-expr with "$true$"; for each statement inside the whilestmt, goto 3(c).
      iv) aliasstmt: if there exists any variable declaration involving variables in D, remove the declaration.
      v) proccall: if there exists any parameter in the proccall involving variables in D, replace the parameter with a nondeterministic value; for each statement in the procedure, goto 3(c).
      vi) clearstmt, putstmt, errorstmt: if the stmt involves variables in D, remove the stmt.
      vii) assertstmt: for the expr inside assertstmt, if there exists any sub-expr involving variables in D, replace the sub-expr with "$true$".
      viii) returnstmt: for the expr in the returnstmt, if there exists any sub-expr involving variables in D, replace the sub-expr with a nondeterministic value in the type of the sub-expr.

Advanced Unbounded Model Checking Based on AIGs, BDD Sweeping, And Quantifier Scheduling

Florian Pigorsch, Christoph Scholl, and Stefan Disch
Albert-Ludwigs-Universität Freiburg, Institut für Informatik,
D-79110 Freiburg im Breisgau, Germany
Email: {pigorsch, scholl, disch}@informatik.uni-freiburg.de

***Abstract*— In this paper we present a complete method for verifying properties expressed in the temporal logic CTL. In contrast to the majority of verification methods presented in recent years, we support *unbounded* model checking based on symbolic representations of characteristic functions. Among others, our method is based on an advanced And-Inverter Graph (AIG) implementation, quantifier scheduling, and BDD sweeping. For several examples, our method outperforms BDD based symbolic model checking by orders of magnitude. However, our approach is also able to produce competitive results for cases where BDD are known to perform well.**

I. Introduction

Given a sequential circuit and properties in some temporal logic like CTL or LTL, model checking is a method for verifying these properties [1], [2]. In the early nineties, by introducing *symbolic* model checking, Burch et al. substantially extended the class of systems which can be verified [3], [4]. In symbolic model checking binary decision diagrams (BDDs) [5] are used both for state set representation and for state traversal. Sets of states are represented by characteristic functions which in turn are represented by BDDs.

However, in the last few years SAT based techniques like Bounded Model Checking (BMC) [6], [7] have been attracting much interest, since industrial needs ask for methods avoiding the well known memory explosion problem which may occur during symbolic model checking of large circuits. BMC applied to certain properties (invariants or, more generally, LTL formulas) 'unfolds' the transition relation for k steps in order to find counterexamples. If no counterexample of length k is found, then k is increased and BMC is used again. For *proving* properties using BMC a suitable upper bound on k is needed. In the case of invariants, e.g., the search for counterexamples can be stopped, when k equals the *diameter* of the system, i.e., the maximum length of all shortest paths between states in the system. Then, BMC ends up with a proof of the property. Unfortunately, computing diameters of large systems turns out to be hard. The problem may be reduced to the validity check of a quantified Boolean formula (QBF) with alternating existential and universal quantifiers [6]. Since this check is usually hard for large systems, BMC is mostly used as an incomplete method for finding errors in practice.[1]

In this paper, we present a *complete* method for verifying properties expressed in the temporal logic CTL. Our method is based on a symbolic representation of sets of states. However, our symbolic representation relies on And-Inverter Graphs (AIGs) [9], [10] instead of BDDs. So far, And-Inverter Graphs have been successfully applied in combinational equivalence checking [9], [10] and in BMC for simplifying representations of transition relations [11]. Basically, they are Boolean circuits which consist of AND gates and inverters only. In contrast to BDDs, AIGs do not provide canonical representations of Boolean functions. Since we do not need canonical representations for representing sets of states, we are able to avoid memory blow-ups during the construction of (canonical) BDDs. On the other hand, checks for satisfiability or validity, which are needed during the model checking process, do not come for free as for BDDs, because there are different AIG representations for constants 0 and 1.

In order to obtain as much sharing of subcircuits as possible we make use of a special version of AIGs, the so-called functionally reduced AIGs (FRAIGs) which were introduced by Mishchenko et al. [12] in the context of logic synthesis, technology mapping and combinational equivalence checking. Like general AIGs, FRAIGs still form non-canonical representations of Boolean functions, but they have the additional property that they do not contain any pair of functionally equivalent nodes. This invariant is maintained during construction of FRAIGs by using a SAT solver. In addition, the construction of FRAIGs is assisted by functional simulation in order to avoid unnecessary SAT checks for pairs of nodes for which already simulation is able to prove non-equivalence. Similar ideas for compressing AIG representations using 'SAT sweeping' and functional simulation can be also found in [11].

The most difficult step during model checking using FRAIGs is the elimination of existential quantifiers. As in [13], [14], [15] existential quantifiers $\exists x f$ are eliminated by replacing them by $f|_{x=0} + f|_{x=1}$. Of course, in the worst case the elimination of one quantifier may double the size of the representation. Although it is not very likely that this worst case behavior can be avoided in random examples (since SAT checking is NP hard), we show in our experimental results that we succeed in limiting the increase in size by several measures including a clever choice of the order of quantifications ('quantifier scheduling'). Interestingly, in contrast to a widespread belief [16], [17], [18] our results prove that – for our approach – quantifier elimination by a circuit-based computation of $f|_{x=0} + f|_{x=1}$ is not restricted to models with a small number of inputs (which have to be quantified during symbolic model checking). Our novel method for quantifier scheduling is based on estimations on the AIG sizes of the results after performing quantifier elimination. In Section V we motivate the importance of quantifier schedu

[1] Another possibility consists in increasing k up to the length of the longest simple path between two states [8]. Whereas it is easier to determine the length of the longest simple path than to determine the diameter of the system, the longest simple path may be exponentially longer than the diameter. If this is the case, unfolding the transition relation for such a large number of steps will be prohibitive.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

giving an example and we describe the approach in more detail. Note that our way of eliminating quantifiers ($\exists x f = f|_{x=0} + f|_{x=1}$) also motivated the use of *functionally reduced* AIGs (FRAIGs) instead of 'standard' AIGs: Since a trivial implementation of quantifying several input variables would lead to an exponential growth of the representation, we need the more aggressive form of enforcing sharing of subcircuits which is provided by FRAIGs.

Other techniques for limiting the sizes of our representations of state sets are node selection heuristics and BDD sweeping:

- Whenever a new node is inserted into our FRAIG representation, we check whether there is already a node in the representation which is functionally equivalent to this new node (using SAT combined with simulation). If there is already a functionally equivalent node, we keep only one representation for the function and replace the representation of one node by the other (this is in contrast to [12] where various representations of the same function are kept for technology mapping purposes). In order to keep the overall size of the representation small we have to select carefully which representation is kept (see Section IV).
- BDD sweeping is known from combinational equivalence checking [9], [10] and builds BDDs for AIG nodes starting at the primary inputs until a certain node limit is reached. BDD sweeping is used there from time to time in order to identify equivalent nodes in the AIG. Since we are using SAT for maintaining the FRAIG invariant we do not need BDD sweeping with this objective. In contrast to the traditional use of BDD sweeping we make use of BDD sweeping in the cone of selected output functions of our FRAIG representation in order to compute smaller AIG representations. After one step of BDD sweeping we check whether our FRAIG representations decrease in size when parts of the FRAIG representation are replaced by subgraphs which are structurally equivalent to the BDDs computed during BDD sweeping.

Interpolation based model checking [17] is related to our approach in the sense that it also provides a method for unbounded model checking. In contrast to our approach [17] does not handle CTL properties, but invariants, and it does not use exact image computations, but overapproximations by so-called Craig interpolants. Due to the overapproximated image computation the method of [17] needs to be applied iteratively on unfoldings of the transition relation for an increasing number of steps (as in Bounded Model Checking). Our method does not need several unfoldings, but it can be used in standard symbolic model checking just replacing BDD representations for state sets by AIG based representations. Other related approaches perform quantifier elimination by using a SAT solver for enumerating all satisfying assignments of a given function [16], [19]. During the enumeration process disjunctions of cubes (or conjunctions of clauses) are collected leading to a two-level representation of the result of the quantification. Characteristic functions for sets of states and transition relations are expressed in conjunctive normal form (CNF) limiting the method to functions having efficient two-level representations. The idea of SAT-based quantifier elimination was refined in [18]. Whereas this method is still based on enumerations of satisfying assignments of a function f, disjunctions of cubes are replaced by disjunctions of cofactors of the function f.

The following novel contributions are introduced by our approach:

- We developed methods for quantifier scheduling which are especially tailored towards our state set representations using FRAIGs. We can show that a proper scheduling of quantifications can lead from exponential representations to representations of linear size.
- The size of the FRAIG representations is limited by heuristics for node selection when functionally equivalent nodes are identified.
- We are using BDD sweeping as a method for non-local logic optimization of our FRAIG representations. BDD sweeping is controlled by heuristics based on the size of the AIG representations and on the success of previous runs of BDD sweeping.

We applied our representations of state sets and of transition functions to CTL model checking. We are using a standard CTL model checking algorithm based on symbolic representations of state sets. However, we make use of degrees of freedom in CTL model checking by preferring operations which are beneficial for our representation (see also Section II).

Our experimental results prove the efficiency of our approach. For several examples, our method outperforms BDD based symbolic model checking by orders of magnitude. However, note that our approach is also able to produce competitive results for cases where BDDs are known to perform well (which was not observed for approaches [13], [14], e.g.). We show in detail how our concepts such as quantifier scheduling, node selection heuristics and BDD sweeping as a non-local optimization step contribute to the success of our experiments.

The paper is structured as follows: We begin with a brief review of CTL model checking in Section II. Then we describe both And-Inverter Graphs (AIGs) in general and the special version of AIGs we use as a data structure for model checking (Section III). In Section IV we describe our heuristics for node selection and in Section V we present our method for quantifier scheduling. AIG compression techniques by BDD sweeping are given in Section VI. After presenting experimental results in Section VII we give some conclusions and future directions in Section VIII.

II. Preliminaries

We use our FRAIG representation in the context of symbolic model checking [3], [4].

Symbolic model checking is applied to Kripke structures (which may be derived from sequential circuits) on the one hand and to a formula of a temporal logic (in our case CTL (Computation Tree Logic)) on the other hand.

An essential step in the recursive evaluation of CTL formulas is the preimage computation which computes for a set of states $Sat(\phi)$ the set of states $Sat(EX\phi)$ with at least one successor in $Sat(\phi)$:

$$\chi_{Sat(EX\phi)}(\vec{q},\vec{x}) := \exists \vec{q}' \exists \vec{x}' \Big(\chi_R(\vec{q},\vec{x},\vec{q}') \cdot \big(\chi_{Sat(\phi)}|_{\substack{\vec{q}\leftarrow\vec{q}' \\ \vec{x}\leftarrow\vec{x}'}}\big)(\vec{q}',\vec{x}') \Big) \quad (1)$$

(As usual χ_M means the characteristic function of set M, $\vec{x}$ represents the current input variables, $\vec{q}$ the current state variables, $\vec{q}\,'$ the next state variables, and $\vec{x}\,'$ the next input variables. χ_R represents the transition relation of the Kripke structure.)

It is well known that the same formula can also be computed based on transition *functions* δ_i of the sequential circui

of the transition *relation* R:

$$\chi_{Sat(EX\phi)}(\vec{q},\vec{x}) := \exists \vec{x}' \Big(\chi_{Sat(\phi)} | \begin{array}{c} q_1 \leftarrow \delta_1(\vec{q},\vec{x}) \\ \dots \\ q_m \leftarrow \delta_m(\vec{q},\vec{x}) \\ \vec{x} \leftarrow \vec{x}' \end{array} \Big)(\vec{q},\vec{x},\vec{x}') \Big) \quad (2)$$

In our implementation of the model checking procedure we always prefer Equation (2) over Equation (1), since the substitution operation is easy in the AIG context and can be performed in parallel for several substitutions. Although we use sophisticated methods to prevent memory blow-ups due to quantification, in principle quantification needs special attention, since quantifying a single variable has the risk of doubling the size of the representation. If not needed, we do not take this risk and we avoid the additional effort of preventing the representation from increasing.

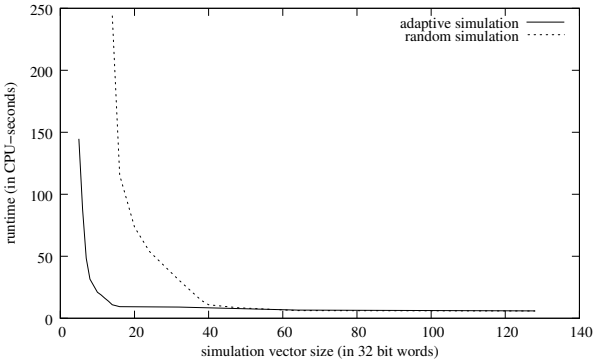


Fig. 1. Impact of adaptive simulation (picojava/icu benchmark)

III. And-Inverter Graphs

Recently, And-Inverter Graphs (AIGs) [9], [10] enjoy a widespread application in combinational equivalence checking and Bounded Model Checking (BMC). They are simply a special kind of directed acyclic graphs representing boolean functions. There are three types of nodes: *and nodes* with two outgoing edges, modeling the Boolean conjunction of the functions represented by the two edges, *variable nodes* with no outgoing edges but labelled with a variable name, representing boolean variables, and a special terminal node with no outgoing edges, forming the constant 0 function.

The edges of an AIG may contain negation marks that denote complementation.

Constructing AIGs using one level structural hashing [10] assures that we do not have two different nodes with the same pair of successors.

A. Functionally Reduced And-Inverter Graphs

AIG representations of Boolean functions are not canonical – for each Boolean function there exist many structurally different AIGs. Actually an AIG may contain functionally redundant nodes, i.e., nodes which are roots of structurally different subgraphs representing the same functions.

Redundant nodes lead to two problems: On the one hand, the graph structure is inefficient. Redundant nodes could be merged to reduce the graph size. On the other, checking the equivalence of two nodes needs additional effort.

To address these problems Mishchenko et al. [12] introduced the notion of functionally reduced AIGs (FRAIGs). The main idea is to check for equivalent nodes using SAT-based equivalence checking techniques while constructing an AIG and to merge them immediately. (In a similar approach Kuehlmann [11] uses 'SAT sweeping' from time to time in order to remove functionally equivalent nodes in AIGs which were not reduced immediately during construction.) This approach establishes the *functional reduction property*: There will not be any two nodes in an FRAIG representing the same Boolean function (and there will not be a pair of nodes where one represents the complement of the Boolean function represented by the other).

B. An AIG Package for Model Checking

Since we use our AIG package for state set representations in CTL model checking, we have different requirements compared to usual packages for combinational equivalence checking. In this section we have a brief look at the key features of our package.

Apart from standard Boolean operations which are translated into AND operations and / or complementations we have to support substitution and existential quantification. Substitution of variables by functions is basically reduced to replacements of inputs of an AIG by subgraphs representing these functions and it can be easily performed for several variables in parallel. Existential quantification $\exists x f$ is reduced to $f|_{x=0}+f|_{x=1}$ (with optimizations described in the following sections).

Whereas in combinational equivalence checking only insertion of nodes has to be supported, we need efficient methods for the deletion of nodes. Nodes have to be deleted when certain state set representations are not needed any longer during the model checking procedure and when functionally equivalent nodes are merged into one representation. The hash table we use for one-level structural hashing (applied as a fast technique for detecting isomorphic AIG nodes) permits lookups in constant time. We have enriched the data structure by adding linked lists to each AIG node chaining all hash table entries affected by this node. This allows for the fast deletion of all occurrences of a node from the hash table without inspecting all table entries.

To maintain the functional reduction property, we use a simulation guided, SAT based equivalence checking method known from the BMC and combinational equivalence checking domains as proposed in [11], [12]. The idea is to avoid powerful methods for easy problems: If for a given pair of nodes simulation is already able to prove non-equivalence, more time consuming SAT checks are not needed. The simulation vectors are initially random, but they are updated using feedback from satisfied SAT instances. Always maintaining a fixed number of simulation vectors we use a simple FIFO replacement method when new vectors are inserted. Figure 1 shows the impact of different simulation vector sizes and the use of learned simulation vectors in a typical model checking run. Depending on the size of the simulation vectors used, the dashed line shows the run times for the complete model checking run, when the learning of distinguishing simulation vectors from satisfied SAT instances is turned off. The solid line shows the same run times for the case that learning is turned on. At least for the smaller sizes of simulation vectors learning leads to considerable improvements of run times. Obviously, by learning we obtain 'good' simulation vectors which are able to prevent time consuming SAT checks during future node insertions.

Additional features of our AIG package which are used during model checking are node selection, quantifier scheduling, and BDD sweeping. They will be described in the following three sections.

IV. Node Selection in Case of Equivalence

When constructing a new node in an AIG, one will often encounter the situation that the current AIG already contains a functionally equivalent node which is the root of a structurally different subgraph. Since the functional reduction invariant must be maintained, only one of the two nodes can be kept and the other one needs to be removed from the AIG. Unlike the approach in [12] where the already existing AIG node is kept and all equivalent nodes are stored in a list of possible structural representations for later technology mapping, our AIG package tries to keep the memory consumption as low as possible and thus destroys redundant nodes. This strategy is vital for a successful employment of AIGs in the model checking domain.

The question is whether to preserve the old, existing node or the newly created one. We use two different heuristics to conquer the problem:

- h_{keep}. We always keep the old node and discard the new node. The drawback of this trivial method is the possible rejection of more efficient structural representations.
- h_{size}. We keep the node that structurally depends on less variables. If the nodes have equal support sizes, we consider the subgraphs (cones) rooted by the two nodes and select the node which has a smaller cone. [2]

In our experiments (see Section VII) we will show that the naive node selection heuristics h_{keep} may result in high and even unmanageable node counts, while the more advanced one is able to reduce the AIGs to reasonable sizes.[3]

V. Quantifier Scheduling

During model checking we eliminate existential quantifiers $\exists x f$ by replacing them by $f|_{x=0}+f|_{x=1}$. In the worst case this elimination may double the size of the representation. Thus, after an existential quantification of a series of variables the size of the representation may potentially show an exponential blow-up. In this section we will present a heuristic method which aims at limiting this (potential) increase in size by a clever choice of the order of quantifications ('quantifier scheduling').

A. A Motivating Example

First of all, we give a motivating example which shows that the order of quantifications may be essential for avoiding memory blow-ups. Consider a simple carry ripple adder which computes the sum $(s_n, \ldots, s_0)$ for two operands $(a_{n-1}, \ldots, a_0)$ and $(b_{n-1}, \ldots, b_0)$. Now we want to compute the set of inputs $(b_{n-1}, \ldots, b_0)$ with the property that there is an input $(a_{n-1}, \ldots, a_0)$ with $2^{n-1} \leq \sum_{i=0}^{n-1} a_i 2^i + \sum_{i=0}^{n-1} b_i 2^i < 2^n$, i.e. with $s_n = 0$ and $s_{n-1} = 1$. The problem may be solved by computing a symbolic (BDD or AIG based) representation of $\overline{s_n} \cdot s_{n-1}$ based on the carry ripple circuit and by computing the existential quantification $\exists a_{n-1} \ldots \exists a_0 \overline{s_n} \cdot s_{n-1}$. The result of the quantification is a representation of a characteristic function for the set of inputs $(b_{n-1}, \ldots, b_0)$ fulfilling the given property. Since it is easy to see that this set of inputs is equal to $\mathbb{B}^n$, the final result has to be equal to 1. Now we consider the two extreme cases for the order of quantification: The first order UP is $(a_0, \ldots, a_{n-1})$ (i.e. we start with the quantification of the least significant bit) and the second order $DOWN$ is $(a_{n-1}, \ldots a_0)$ starting with the quantification of the most significant bit. Remember that we reduce quantification wrt. one variable to the disjunction of positive and negative cofactors. Figure 2.(1) shows the result of the quantification wrt. the first variable a_0 of order UP (for simplicity applied to the given carry ripple circuit, not to the corresponding AIG, which has roughly the same structure). The illustration shows that the propagation of constants 0 and 1 for the negative and the positive cofactor wrt. a_0 already stops at bit position 1 and no subcircuit sharing can be observed in the remaining circuit. Thus, the size of the circuit is almost doubled by quantification.

However, if we quantify variable a_{n-1} first, we obtain the situation shown in Figure 2.(2). In this case most parts of the circuit are shared between the positive and the negative cofactor. Since we use FRAIGs which identify functionally equivalent nodes, the corresponding FRAIG also shows this sharing. The duplication of the number of nodes as observed in the previous case can not be seen here. The effect shown above continues during the following quantifications according to the orders UP and $DOWN$: As shown in Table I for the example of a 14-bit-adder, we observe an exponential blow-up of AIG nodes during quantification according to order UP, whereas the number of AIG nodes is monotonically decreasing for quantification order $DOWN$. (Note that BDD sweeping described in the next section was not used in this experiment.) Altogether our example shows that there may be an exponential gap wrt. AIG sizes between good and bad orders of quantification. So we have a strong need for heuristics computing good quantification orders.

B. A Heuristic Approach to Quantifier Scheduling

Here we present a method for quantifier scheduling which is especially tailored towards our state set representations using FRAIGs. Our greedy method is based on estimations on the sizes of the results after performing quantifier elimination. Before performing a quantification $\exists x f = f|_{x=0} + f|_{x=1}$ for some variable x and some function f represented by a FRAIG, we compute an estimate on the FRAIG size of the final result:

- In a first step we consider the subgraph of the AIG representing f and for each $\epsilon \in \{0, 1\}$ we determine by two traversals of this subgraph the set R_ϵ of nodes which are not removed by propagation of constant ϵ.
- In a second step we compute an estimate for the node sharing between the representations of the positive and the negative cofactor: If a node k which occurs both in R_0 and R_1 is not connected to variable x by a path in the AIG graph, then it does not depend on x and thus the nodes corresponding to k in the representation c

[2] It is easy to see that h_{size} never replaces a node k by another node k' having k in its cone (which would create a loop in the AIG).

[3] In the case the used heuristics suggest to keep the old node, the only thing to do is to delete the new node. But if the new node is selected, we use a technique similar to implementation techniques known from BDD packages: All edges of the AIG pointing to the old node must be modified to reference the new node. Since the data structure used in our AIG package does not provide an efficient method for finding all predecessors of a node, we need to use a more subtle replacement method: we actually transfer the data of the new node object into the old node object and then delete the new node. By doing this no edge has to be touched.

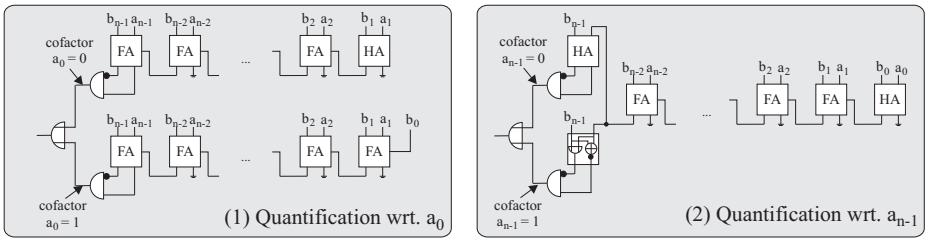


Fig. 2. $\overline{s_n} \cdot s_{n-1}$ after (1) quantification of a_0 and (2) quantification of a_{n-1}.

TABLE I

14-BIT-ADDER, FUNCTION $\overline{s_{14}} \cdot s_{13}$: NUMBER OF AIG NODES AFTER QUANTIFICATIONS OF SINGLE VARIABLES a_i, ORDERS UP AND $DOWN$

Quant. Nr.	orig.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
UP	111	105	106	115	140	197	318	567	1072	2089	4130	8219	16404	32781	1
DOWN	111	105	99	93	87	81	75	69	63	57	51	45	39	33	1

and $f|_{x=1}$ are the same (due to the functional reduction property). So the set $R_{0,1}$ of all such nodes is our estimate for the set of nodes shared between $f|_{x=0}$ and $f|_{x=1}$.[4]

- Finally, our estimate for the size of $f|_{x=0} + f|_{x=1}$ is $1 + |R_0| + |R_1| - |R_{0,1}|$.

For reasons of efficiency our estimate does not consider node sharing between nodes for both cofactors on the one hand and nodes which are not in the subgraph representing f on the other hand. Additionally, possible restructuring of the AIG during node insertion (see Sections IV and VI) is not taken into account. However, as shown by experimental results, our heuristic estimate seems to be reasonable for computing a good order for quantification: Whenever a number of variables $x_1, \ldots, x_n$ has to be quantified for a function f, we compute for each x_i our estimate of the size of $\exists x_i f$ and (greedily) start with quantification of the variable with smallest cost. Then the method is repeated to determine the next variable to be quantified and so on.

We would like to point out that in the case of our motivating example from above our heuristic method leads to quantification order $DOWN$ which produces a monotonically decreasing number of AIG nodes, whereas unfavorable orders like the order UP shown above lead to an exponential peak size in the number of AIG nodes before the final result 1 is computed.

VI. BDD SWEEPING

BDD sweeping [20], [9], [10] is a well-known technique from the domain of combinational equivalence checking (CEC). It builds BDDs for AIG nodes starting at the primary inputs until a certain node limit is reached. Whereas in [20], [9], [10] BDD sweeping is used in order to identify functionally equivalent nodes in the AIG, this is not needed in our case, since we always maintain the functional reduction property using SAT as described in Section III. Here we use BDD sweeping as a means for non-local optimizations of our AIG representations. However, for reasons of efficiency both the number of BDD sweepings and the cost of a single BDD sweeping have to be limited.

From time to time, after certain operations of the AIG package, BDD sweeping is applied to the cone of the corresponding result. BDD sweeping builds a BDD for the cone of the given AIG node starting from the variable nodes and using AND and NOT operations. Variable reordering applied by the BDD package automatically tries to find an optimal variable order in terms of BDD node count. If BDD sweeping is able to compute the BDD for the given AIG node, then we check whether it makes sense to replace the cone of the AIG node by an AIG which is structurally equivalent to the BDD. Here we exploit the fact that any BDD node can be interpreted as a multiplexer, which can be transformed into an AIG with exactly three AIG nodes. Thus, if the size of the generated BDD is smaller than one third of the size of the given cone, we create an AIG from the BDD structure by recursively transforming the BDD nodes to their three-node AIG representation. When inserting this new AIG into the AIG package, node selection heuristics as described in Section IV are used as usual with the additional effect that subgraphs of the new AIG may be replaced by smaller (functional equivalent) representations which are already present in the existing AIG graph.

In order to limit the *cost* of a single BDD sweeping we use a variable *BDD_limit*. Whenever the number of nodes in the BDD package is larger than *BDD_limit*, the BDD construction is aborted.

In order to limit the *number* of BDD sweepings, we decided to confine BDD sweeping to the results of cofactor operations which occur during existential quantification, since existential quantification of a variable is the only operation that has the risk of doubling the size of the representation. Moreover, BDD sweeping is not applied after *all* cofactor operations, but only after a small fraction of cofactor operations controlled by sophisticated heuristics based on the sizes of the results and the success of previous BDD sweepings. To avoid unnecessary BDD sweepings, BDD sweeping is only applied, if the AIG size of the current operation is larger than a some variable *AIG_limit*. *AIG_limit* is initialized to a certain constant (100 in our current implementation) and it evolves as follows:

- If a BDD is successfully built within the node limit *BDD_limit*, but it is not used in the AIG due to its size, *AIG_limit* is multiplied by a certain factor $f_1 > 1$ ($f_1 = 1.2$ in our current implementation).
- If the BDD construction is aborted, since *BDD_limit* is exceeded, *AIG_limit* is multiplied by some larger factor $f_2 > 1$ ($f_2 = 4$ in our current implementation).
- If a BDD is successfully built and a structural equivalent AIG is inserted into the AIG package, then *AIG*

[4] Situations where a node does not functionally depend on a variable x, but is (structurally) connected to x, may be possible in AIGs due to non-canonicity. However they are neglected for reasons of efficiency.

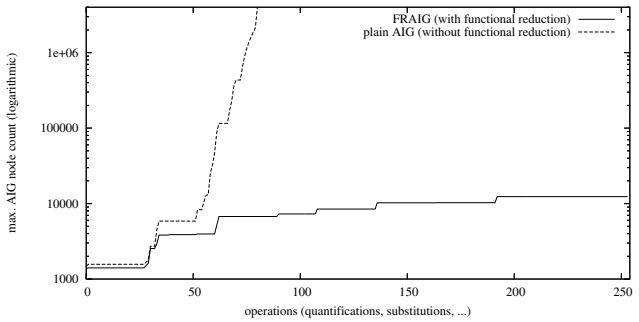


Fig. 3. Impact of functional reduction (picojava/icu).

set to the size of the resulting AIG.

- Whenever BDD sweeping is not applied, since *AIG_limit* is too large, *AIG_limit* is decreased by multiplication with $1 - f_3 \cdot (\frac{1}{2})^{abort}$, where f_3 is a small constant ($0 < f_3 < 1$) and $abort$ is the number of times the BDD construction is aborted due to an exceeded node limit (in our implementation we used $f_3 = 0.01$).

The presented heuristics ensure that unsuccessful BDD sweeping runs result in fewer BDD sweeping runs in the future.

The variable *BDD_limit* for aborting BDD constructions has to be high enough to allow for BDD variable reordering and is set to $\max(100 \cdot \textit{AIG\_limit}, 10^6)$ in our implementation.

Whenever BDD sweeping is aborted, it ends up with a number of BDDs for AIG nodes in the considered cone, since it computes BDDs beginning with the input variables of the cone. The procedure described above can be easily extended by making use of the BDDs computed so far. However, this feature is not yet realized in our prototype implementation.

VII. Experimental Results

We performed a number of experiments for evaluating our approach which we called '*AIG-MC*'. The examples AM2910, Coherence, DAIO, Picojava/ICU, Viper, and Barrelshifter are taken from the *VIS Verification Benchmark* set [21].[5] For each benchmark we checked all CTL formulas provided in the *VIS Verification Benchmark* set. We randomly selected the benchmarks from the set of those benchmarks in the *VIS Verification Benchmark* set which have CTL specifications. Since the prototype implementation of our model checker is currently not yet able to read the benchmark format used in VIS, we had to translate the hierarchical and multi-valued models into flat, binary encoded models. This included a manual adaption of the CTL formulas to the new binary encoding variables.[6] Moreover, we used a pipelined ALU ('PALU') similar to the one presented in [3]. The pipelined ALU includes 16 registers, a combinational adder, a combinational multiplier, and bitwise operations.[7] As in [3] the inputs to the ALU are instruction codes containing a specification of the operation, the source registers and the destination register. For the pipelined ALU we checked the CTL formula $\phi = AG\big(''\mathtt{R}_2 := \mathtt{R}_0 \oplus \mathtt{R}_1'' \rightarrow \big((AX)^2\mathbf{R}_0 + (AX)^2\mathbf{R}_1 \equiv (AX)^3\mathbf{R}_2\big)\big)$

[5]The number of registers in the barrelshifter was increased from 4 to 10.

[6]The complete set of benchmarks is provided in [22].

[7]The bit width of all operations and registers for palu12,4 is 12, for palu14,4 14 and for palu16,4 16, respectively.

TABLE II
Node selection heuristics in *AIG-MC*

benchmark (ctls)	h_{keep}		h_{size}		
	nodes	time	nodes	repl.	time
am2910 (1-3,5-6)	5354	5.9	2129	232	0.9
barrel10,4 (1)	5918	39.6	5918	0	37.2
coherence (1,7)	1310	0.3	1303	20	0.3
daio (1-4)	50862	10680.9	17069	13078	245.0
decay11	15578	33.6	15578	1	33.7
palu12,4 (xor)	3802	2.1	3802	216	1.9
palu14,4 (xor)	4654	2.7	4654	252	2.6
palu16,4 (xor)	5586	3.6	5586	288	3.3
picojava/icu (1)	23924	21.6	10650	937	8.5
viper (1-3)	15975	7.7	15970	69	11.7

(similar to formula (1) from [3]).[8] The benchmarks named 'decay$n$' contain registers of bit widths n and they compute decaying sums of sequences of inputs according to the formula $register_{new} = \lceil \frac{register_{old}}{2} \rceil + input$.[9]

Note that the barrelshifter example used here is different from the barrelshifter example given in [13], [14]. The examples in [13], [14] do not contain inputs, and thus, quantification is not needed during the fixed point computation of the model checking procedure (see Section II, equation (2)). We did not compare our results to the results from [13], [14], since our goal was to prove that we are able to handle quantification as well. In this sense our experiments show that the objection raised by McMillan ([17], Section 1.1) 'because of the expense of quantifier elimination, this approach is limited to models with a small number of inputs (typically zero or one)' does not apply to our approach.

All experiments were performed on a 2 GHz Dual-Opteron workstation running Debian Linux. We used a timeout of 12 CPU hours.

First of all we demonstrate the effect of functional reduction by means of a typical example in Fig. 3. Fig. 3 shows the number of AIG nodes which are needed during model checking of the picojava/icu benchmark. In this experiment we turned off quantifier scheduling and BDD sweeping in order to concentrate on the effect of functional reduction. The numbers of AIG nodes were recorded after each quantification of a variable and after each substitution, thus the x-axis represents the ongoing progress of the model checking procedure. The numbers of nodes are presented with a logarithmic scale. The dashed line shows the number of nodes which are needed when functional reduction using SAT is turned off, the solid line shows the number of nodes of our FRAIG package using SAT based functional reduction. The example clearly shows that functional reduction is essential for the success in this kind of applications. An AIG package only using structural hashing is not able to provide sufficient compaction. For this reason we always consider results using our FRAIG package with SAT based functional reduction in the following.

In the first experiment we evaluated the effect of our node selection heuristics from Section IV. Table II lists the peak node counts and run times in CPU seconds for the two different proposed node selection heuristics: the naive method h_{keep} (always keeping the already existing node) and h_{size}. For h_{size} Table II also reports the numbers of node replacement

[8]Given an *exor* operation in the instruction register the formula basically checks whether the contents of the destination register in three steps are the same as the *or* operation of the contents of the operand registers in two steps. This would be true for an *or* operation in the instruction register, but is obviously not true for the *exor* operation.

[9]The property asks whether there is a sequence of inputs such t[illegible] binary number R in the register $2^{n-1} \leq R < 2^n$.

TABLE III

AIG-MC W/O, WITH QUANT. SCHEDULING, BDD-MC, VIS

benchmark	ctl	w/o quant. sched. nodes	w/o quant. sched. time	quant. sched. nodes	quant. sched. time	BDD-MC time	VIS time
am2910	1	1132	0.2	1132	0.2	1.5	3.2
	2	1143	0.3	1143	0.3	1.5	3.2
	3	1605	0.8	1650	0.7	1.5	3.2
	4	16039	91.9	13818	51.2	9.0	5.5
	5	1977	1.1	1880	1.6	1.5	3.2
	6	1205	0.3	1181	0.3	1.5	0.8
barrel10,4	1	5918	50.3	5918	51.2	>12h	>12h
coherence	1	1303	0.3	1303	0.3	0.2	0.4
	2	43285	334.3	49010	172.4	1.0	0.4
	3	11744	22.3	10001	13.6	1.4	0.4
	4	25190	72.6	18781	41.7	1.6	0.4
	5	18590	40.5	7895	7.2	2.1	0.5
	6	23814	176.4	25298	88.3	0.8	0.4
	7	1303	0.3	1303	0.3	0.2	0.4
	8	101185	609.8	42042	151.6	37.8	0.4
	9	18590	33.1	7895	5.0	1.6	0.4
daio	1	996	0.6	996	0.6	0.1	0.3
	2	996	0.8	996	0.8	0.2	0.4
	3	1768	1.4	1768	1.4	0.3	0.4
	4	996	0.8	996	0.8	0.2	0.4
decay32	1	1901	6.3	718	1.2	0.5	0.0
decay48	1	2814	30.3	1070	4.0	2.8	0.2
decay64	1	3736	83.0	1422	9.8	21.1	0.3
palu12,4	xor	3802	153.4	3832	76.2	>12h	>12h
palu14,4	xor	4654	59.0	4684	119.4	>12h	>12h
palu16,4	xor	5586	1062.6	5616	91.1	>12h	>12h
picojava/icu	1	8144	6.6	2869	5.0	1.0	2.0
viper	1	15757	3.1	15757	3.0	43.4	75.0
	2	15757	6.7	15757	6.1	43.3	73.0
	3	15757	3.1	15757	3.0	43.3	74.1

TABLE IV

DETAILED RESULTS FOR *AIG-MC*

benchmark	bdd sweeping applic.	bdd sweeping succ.	bdd sweeping limit	sat applic.	sat equiv
am2910	0.06%	59.80%	0%	5.00%	45.58%
barrel10,4	0.01%	0%	100%	1.90%	42.51%
coherence	0.02%	74.06%	0.37%	0.97%	69.47%
decay32	0.39%	11.11%	0%	37.00%	4.65%
decay48	0.26%	9.09%	0%	55.09%	2.40%
decay64	0.20%	7.69%	0%	72.36%	1.50%
daio	0.12%	98.13%	0%	11.11%	93.99%
palu12,4	0.03%	12.5%	62.5%	1.48%	74.07%
palu14,4	0.02%	12.5%	62.5%	1.28%	76.92%
palu16,4	0.02%	12.5%	62.5%	1.12%	78.28%
picojava/icu	0.07%	33.33%	0%	5.06%	31.47%
viper	0.01%	33.33%	0%	4.56%	60.49%

steps. The numbers for each benchmark are averaged over all different CTL formulas. In this experiment we turned off BDD sweeping, because the naive method h_{keep} would never use the results of BDD sweeping. (Thus the results would be biased towards h_{size}, since it can exploit BDD sweeping whilst h_{keep} does not profit from it.) In the comparison we omitted the results for formula 4 of example 'AM2910' and formulas 2-6 and 8-9 of example 'Coherence', since the computation did not finish for h_{keep} within our limit on CPU time. (For formulas 2-6 and 8-9 of 'Coherence' the computation without BDD sweeping did not finish for h_{size} as well, i.e., BDD sweeping is essential for success with this benchmark (see also experiments of Table III).)

The results clearly show that the node selection heuristics are of great importance for obtaining good results: The heuristics h_{size} lead to a considerable decrease in peak node counts. The most impressive examples are AM2910 and DAIO where the peak number of AIG nodes for the naive method are by a factor of 2.5 and 3.0 higher than for h_{size}. Not only the node counts, but also the run times are greatly reduced by h_{size}, in the case of Picojava from 21.6 CPU seconds with h_{keep} to 8.5 CPU seconds with h_{size}, in the case of DAIO even from 3 CPU hours with h_{keep} to about 4 CPU minutes with h_{size}.

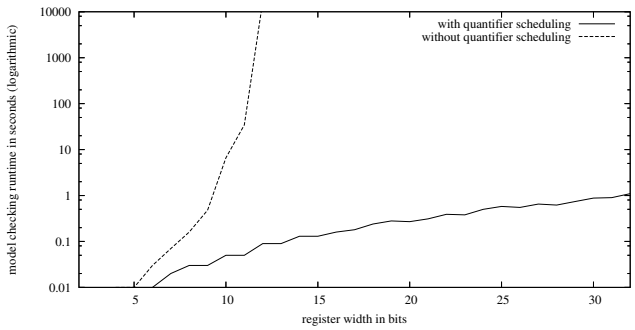


Fig. 4. Example decay$n$ with and without quantifier scheduling, run times.

Since the node selection heuristics h_{size} seems to be the best choice, we always use this method in the following.

In the following experiments we consider our method with BDD sweeping turned on. In the second experiment shown in Table III we evaluated the effect of quantifier scheduling and in the third experiment also shown in this table we compared our results to a BDD based version of our model checker and to the BDD based model checker VIS 2.1 [21]. We ran VIS using dynamic variable ordering with the sifting heuristics, don't care optimization via reachability analysis was turned off.[10]

For the different benchmarks and CTL formulas columns 3 and 4 show the peak node counts and run times for the unmodified quantification order ('w/o quant. sched.') and columns 5 and 6 give the same results for our quantifier scheduling heuristics from Section V. It can be observed that quantifier scheduling improves the peak node counts in most cases; for Coherence, formula 8, e.g., by a factor of 2.4, for decay64 by a factor of 2.6, for Picojava, by a factor of 2.8. The run times are always better or at least in the same range, for example Coherence, formula 8, run times are even reduced from 10.2 CPU minutes to 2.5 CPU minutes.[11]

In Fig. 4 we consider benchmark decay$n$ and have a closer look at the effect of quantifier scheduling: In this case we turned off BDD sweeping in order to perform a separate analysis of the effect of quantifier scheduling: Fig. 4 gives the CPU times for decay$n$ with increasing bit width n both for the version without quantifier scheduling (dashed line) and for the version with quantifier scheduling (solid line) (presented with a logarithmic scale). The experiment shows that without quantifier scheduling the run times grow exponentially with increasing bit width rendering completion of model checking for larger bit widths impossible. (The peak numbers of nodes are not presented here, but they show the same exponential growth.) With quantifier scheduling we observe only a moderate growth of node counts and run times.[12]

[10]For all experiments considered here the default option of performing a reachability analysis before backward model checking gave inferior results. For benchmark am2910 we even observed timeouts (larger than 12 CPU hours) for 5 out of 6 CTL formulas when reachability analysis was turned on.

[11]For examples palu$n$,4 the run times are somewhat misleading (increase of run times for palu14,4 by a factor of 2.0, decrease of run times even by a factor of 11.7 for palu16,4 for the version with quantifier scheduling): A more detailed analysis showed that the run times are almost exclusively due to unsuccessful runs of BDD sweeping (not leading to node replacements in the AIG). Without BDD sweeping the run times remain in the range of a few seconds.

[12]In Table III quantifier scheduling outperforms the version without quantifier scheduling for decay32, decay48, and decay64, but an exponential growth of node counts and run times is not observed. In this experin sweeping was turned on and it was able to prevent the exponentia

The last two columns of Table III give a comparison of our results to our own model checker BDD-MC with FRAIGs replaced by BDDs and to the BDD based model checker VIS, respectively. For barrel10,4, palu12,4, palu14,4, and palu16,4 neither BDD-MC nor VIS were able to provide a result within the CPU limit of 12 CPU hours. However, these examples did not form a problem for our model checker *AIG-MC* and we could solve them within a few seconds (see column 6 of Table III).

In contrast, for the remaining benchmarks taken from the VIS Benchmark set as well as for decay$n$, BDDs are known to perform well and these examples could be solved quickly by VIS. Even for this class of examples our approach finished in shorter time in 10 out of 26 cases and also for the remaining cases we could observe that our approach succeeded in producing competitive results within a few seconds.

For completeness we give some more details for our experiments with BDD sweeping and quantifier scheduling turned on in Table IV. Here again the numbers are averaged over all formulas. Column 2 shows the number of applications of BDD sweeping divided by the total number of attempts to insert a node into the AIG. Column 3 shows the numbers of successful applications of BDD sweeping (i.e. the numbers of BDD sweepings where the results are really used in the AIG package) divided by the total number of BDD sweepings. And finally, column 4 shows the numbers of aborted BDD sweepings (due to exceeded node limits) divided by the total number of BDD sweepings. BDD sweeping is only applied from time to time in all cases and in cases where BDD sweeping is not very successful (especially for examples 'Barrel' and 'PALU') our heuristics from Section VI work in the sense that unsuccessful BDD sweeping runs result in fewer BDD sweeping runs in the future. Column 5 shows the number of SAT checks divided by the total number of attempts to insert a node into the AIG, column 6 shows the fraction of SAT checks which lead to the result that the compared nodes are functionally equivalent. Although we always maintain the functional reduction property of our FRAIGs, the results show that the assistance of SAT by simulation and structural hashing as described in Section III-B assures that SAT is applied only for a small fraction of all node insertions. Moreover, the high percentage of SAT checks proving functional equivalence of two nodes shows the effectiveness of simulation in avoiding unnecessary SAT checks for nodes which are not equivalent.[13]

VIII. Conclusions and Future Work

We presented an approach to unbounded model checking based on And-Inverter Graphs as a representation of characteristic functions. Several methods such as functional reduction using simulation assisted SAT checks, node selection heuristics, quantifier scheduling, and BDD sweeping contribute to the success of our approach. For many examples, our method outperforms BDD based symbolic model checking by orders of magnitude, whereas it is still able to produce competitive results for cases where BDD are known to perform well. Although the experimental results for our current implementation already appear to be impressive, we believe that there remains room for improvement of the heuristics presented in Sections IV, V, and VI. Certainly, our prototype implementation will also profit from the integration of a number of interesting ideas recently developed for optimizing AIG representations such as DAG-aware circuit compression [23], [24] and advanced rewriting methods [15], [24]. In the future we will investigate whether methods for structural SAT solving [9] will be useful in our context and we will explore whether it sometimes makes sense to switch to lazy methods for AIG compression instead of our eager one.

[13] Benchmarks decay$n$ form an exception to this observation: In this case there are many nodes in the representation which are 'almost equivalent', so that simulation with a fixed number of simulation vectors is not very effective in distinguishing between them, leading to a number of SAT checks not proving functional equivalence.

References

[1] A. Sistla and E. Clarke, "The complexity of propositional linear temporal logics," *Journal of the ACM*, vol. 32, no. 3, pp. 733–749, 1985.

[2] E. Clarke, E. Emerson, and A. Sistla, "Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation*, vol. 98(2), pp. 142–170, 1992.

[4] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[5] R. Bryant, "Graph - based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[6] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Constuction and Analysis of Systems*, ser. LNCS, vol. 1579. Springer Verlag, 1999.

[7] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Design Automation Conf.*, 1999.

[8] M. Sheeran, S. Singh, and G. Stalmarck, "Checking Safety Properties Using Induction and a SAT-solver," in *FMCAD*, ser. LNCS, W. H. Jr. and S. Johnson, Eds., vol. 1954. Springer, 2000, pp. 407–420.

[9] V. Paruthi and A. Kuehlmann, "Equivalence checking combining a structural SAT-solver, BDDs, and simulation," in *Int'l Conf. on Comp. Design*, 2000, pp. 459–464.

[10] A. Kuehlmann, V. Paruthi, F. Krohm, and M. M.K. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification," *IEEE Trans. on CAD*, 2002.

[11] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," in *Int'l Conf. on Computer-Aided Design*, 2004, pp. 50–57.

[12] A. Mishchenko, S. Chatterjee, R. Jiang, and R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," EECS Dept., UC Berkeley, Tech. Rep., 03 2005.

[13] P. Abdullah, P. Bjesse, and N. Een, "Symbolic reachability analysis based on sat-solvers," in *Tools and Algorithms for the Constuction and Analysis of Systems*, ser. LNCS, vol. 1785. Springer-Verlag, 2000.

[14] P. Williams, A. Biere, E. Clarke, and A. Gupta, "Combining decision diagrams and SAT procedures for efficient symbolic model checking," in *Computer Aided Verification*, ser. LNCS, vol. 1855. Springer Verlag, 2000, pp. 124–138.

[15] G. Cabodi, M. Crivellari, S. Nocco, and S. Quer, "Circuit based quantification: Back to state set manipulation with unbounded model checking," in *Design, Automation and Test in Europe*, 2005.

[16] K. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *Computer Aided Verification*, ser. LNCS, vol. 2404. Springer, 2002, pp. 250–264.

[17] ——, "Interpolation and SAT-Based Model Checking," in *Computer Aided Verification*, ser. LNCS. Springer, 2003.

[18] M. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based unbounded symbolic model checking using circuit cofactoring," in *Int'l Conf. on Computer-Aided Design*, 2004, pp. 510–517.

[19] H.-J. Kang and I.-C. Park, "Sat-based unbounded symbolic model checking," *IEEE Trans. on CAD*, vol. 24, no. 2, pp. 129–140, February 2005.

[20] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Design Automation Conf.*, 1997, pp. 263–268.

[21] The VIS Group, "VIS Verification Benchmarks." [Online]. Available: http://vlsi.colorado.edu/∼vis/

[22] F. Pigorsch and C. Scholl, "Collection of benchmarks." [Online]. Available: http://www.informatik.uni-freiburg.de/∼pigorsch/benchmarks.html

[23] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *Int'l Conf. on CAD*, 2004, pp. 42–49.

[24] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting," in *Design Automation Conf.*, 2006, pp. 532–535.

Symmetry Reduction for STE Model Checking

Ashish Darbari
Oxford University Computing Lab
Wolfson Building
Parks Road
Oxford OX1 3QD
Email: ashish@comlab.ox.ac.uk

***Abstract*— In spite of the tremendous success of STE model checking one cannot verify circuits with arbitrary large number of state holding elements. In this paper we present a methodology of symmetry reduction for STE model checking, using a novel set of STE inference rules. For symmetric circuit models these rules provide a very effective reduction strategy. When used as tactics, rules help decompose a given STE property to a set containing several classes of equivalent STE properties. A representative from each equivalence class is picked and verified using an STE simulator, and the correctness of the entire class of assertions is deduced, using a theorem that we provide. Finally inference rules are used in the forward direction to deduce the overall statement of correctness. Our experiments on verifying arbitrarily large CAMs and circuits with multiple CAMs, show that these can be verified using a fixed, small number of BDD variables.**

I. Introduction

Symbolic trajectory evaluation — STE in short, has been successfully applied in several large scale circuit verification tasks [1], [8], [14], however by itself it is not sufficient, specially when verifying circuits with a large number of state holding elements like memories [11]. It is also the case that memory based circuits intrinsically have plenty of structural symmetry and intuition suggests that this could be an important domain specific attribute, that can be used for STE property reduction.

There are two challenges involved in achieving an efficient solution to the problem of symmetry reduction. First is the discovery of symmetry in the circuit, and secondly to work out the methodology of computing the reductions in properties and circuit models. In a recent work [4] we proposed that if the representation of the circuit model is lifted higher to encapsulate the structural, symmetry-rich information, then the discovery of symmetry reduces to type checking. In this paper we address the second challenge which is to design and implement a methodology of computing reductions. We will assume that circuits are modelled using the methodology proposed in [4].

II. Overview of reduction

The reduction approach presented in this paper, is centered around the use of a novel set of STE inference rules, and observing that symmetry in circuit models is mirrored by symmetry in STE properties. Together the two give us a sound methodology for enabling symmetry based reduction for STE model checking. Figure 1 shows the overview of the symmetry based reduction methodology.

The question we ask in a typical property verification task, is whether or not the FSM satisfies ($\models$) the STE property. Rather than trying to feed the STE property directly in an STE simulator[1] to verify it, we decompose the property using the STE inference rules into smaller properties. The reduced STE properties are then partitioned into different equivalence classes, and one representative from each equivalence class is fed into an STE simulator for explicit STE simulations. The partitioning of the properties is based on identifying the names of circuit nodes that belong to a bus in the symmetric inputs, and using these nodes as a basis to perform clustering of decomposed STE properties into equivalence classes.

Symmetry in circuit models is mirrored by symmetry in STE properties. This is formalised by a *soundness theorem*, which is then used to justify the existence of the correctness of the whole class of equivalent STE properties from the correctness of one of its member. Once this is achieved, all we need to do then is to use the STE inference rules in the forward direction and compose the overall statement of correctness about the original STE property. Since the complexity of STE model checking relies on the property, rather than the model, reducing property, reduces the circuit model.

III. STE theory and Symmetry

In this section we present a brief introduction of STE model checking, and formalise the relation between symmetry in circuit models and the logic of STE.

A. STE — a brief introduction

STE [13] is a model checking technique that combines the ideas of *ternary modelling* with *symbolic simulation*. Symbolic trajectory evaluation employs a ternary circuit state model, in which the usual binary values 0 and 1 are augmented with a third value X that stands for an unknown. To represent this mathematically, we introduce a partial order relation $\sqsubseteq$, with $X \sqsubseteq 0$ and $X \sqsubseteq 1$. The relation orders values by information content: X stands for a value about which we know nothing, and so is ordered below the specific values 0 and 1. We can turn the set of ternary values into a complete lattice where every element has a *join* or a least-upper bound ($\sqcup$), and a *meet* or a greatest-lower bound ($\sqcap$), by adding an artificial element - $\top$ (called 'top') to get the set of circuit

[1] We use the STE simulator in Forte.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

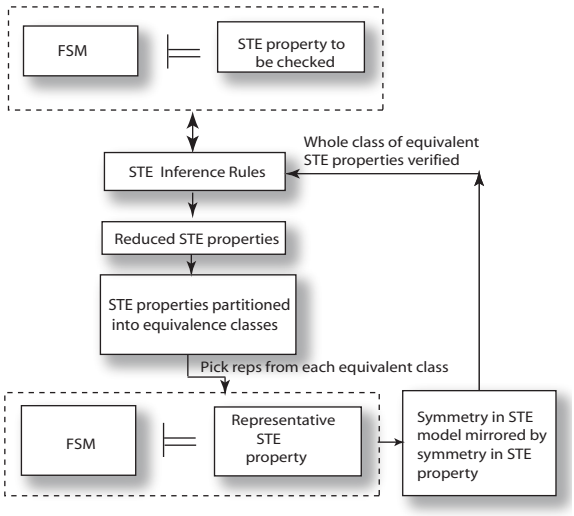


Fig. 1. Overview of symmetry reduction methodology

node values $\{0, 1, \mathrm{X}, \top\}$. The idea of dual-rail encoding is used to denote the four lattice values, by having the concept of a high-rail and a low-rail. Whenever a lattice value is 1 the high rail will take on a Boolean T and the low rail a Boolean F. If the lattice value is 0 the high and low rail are just the inverse of the case for 1. Similarly the other lattice values are defined [10], [13].

In STE, circuit models are represented by a next-state function from lattice states to lattice states. This is also known as an excitation function, and is constructed on-the-fly during simulation, from the netlist description of the circuit. Specifications in STE, take the form of what are known as *symbolic trajectory formulas*. Formally, we define the syntax of formulas [10], [13] as follows:

Definition 1. *Syntax of STE formulas*

$$
\begin{array}{lcll}
f & \triangleq & n \text{ is } 0 & \text{- node } n \text{ has value 0} \\
 & | & n \text{ is } 1 & \text{- node } n \text{ has value 1} \\
 & | & f_1 \text{ and } f_2 & \text{- conjunction of formulas} \\
 & | & f \text{ when } G & \text{- } f \text{ is asserted only when } G \text{ is true} \\
 & | & \mathsf{N} f & \text{- } f \text{ holds in the next time step}
\end{array}
$$

where f_1 and f_2 range over formulas, $n \in \mathit{string}$ ranges over the nodes of the circuit, and G is a propositional formula over Boolean variables (i.e. a Boolean 'function') called a *guard*. The advantage of using a Boolean expression is in specifying conveniently many different operating conditions in a compact form. The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent interdependencies among node values. For example, we can associate an arbitrary propositional formula G with a node using the construct 'n is G' defined by

$$n \text{ is } G \triangleq ((n \text{ is } 1) \text{ when } G) \text{ and } ((n \text{ is } 0) \text{ when } \neg G)$$

We also use a convenient form of expressing the temporal formula, by using *from* and *to* functions. If f is a trajectory formula, without the N operator, and we want to express that f holds from time i to j, we can do it by using the following representation [7]:

$$f \textit{ from } i \textit{ to } j \triangleq \mathsf{N}^i f \text{ and } \mathsf{N}^{i+1} f \text{ and } \ldots \text{ and } \mathsf{N}^{j-1} f$$

where the convention is that $\mathsf{N}^0 f = f$. Semantics of trajectory formulas is usually given with respect to an assignment ϕ of Boolean truth-values to the variables that appear in the guards of the formula.

Definition 2. *Semantics of STE formulas*

$$
\begin{array}{ll}
(\phi, \sigma) \models n \text{ is } 0 & 0 \sqsubseteq \sigma\ 0\ n \\
(\phi, \sigma) \models n \text{ is } 1 & 1 \sqsubseteq \sigma\ 0\ n \\
(\phi, \sigma) \models f_1 \text{ and } f_2 & (\phi, \sigma) \models f_1 \wedge (\phi, \sigma) \models f_2 \\
(\phi, \sigma) \models f \text{ when } G & (\phi \models G) \supset (\phi, \sigma) \models f \\
(\phi, \sigma) \models \mathsf{N} f & (\phi, \sigma_1) \models f
\end{array}
$$

where $\phi \models P$ means the assignment of truth-values given by ϕ satisfies the formula P. The formal definition of $\phi \models P$ is the usual definition for the semantics of propositional formulas. The notation σ_1 denotes the sequence shifted forward by one time point.

The key feature of STE logic is that for any trajectory formula f and assignment ϕ, there exists a unique weakest sequence that satisfies f. This sequence is called the *defining sequence* for f and is written $[f]^\phi$. It is defined recursively as follows:

Definition 3. *Defining Sequence*

$$
\begin{array}{lcl}
[m \text{ is } 0]^\phi\ t\ n & \triangleq & \textit{if } (m{=}n) \wedge (t{=}0) \textit{ then } 0 \textit{ else } X \\
[m \text{ is } 1]^\phi\ t\ n & \triangleq & \textit{if } (m{=}n) \wedge (t{=}0) \textit{ then } 1 \textit{ else } X \\
[f_1 \text{ and } f_2]^\phi\ t\ n & \triangleq & ([f_1]^\phi\ t\ n) \sqcup ([f_2]^\phi\ t\ n) \\
[f \text{ when } G]^\phi\ t\ n & \triangleq & \textit{if } (\phi \models G) \textit{ then } ([f]^\phi\ t\ n) \textit{ else } X \\
[\mathsf{N} f]^\phi\ t\ n & \triangleq & \textit{if } (t{\neq}0) \textit{ then } ([f]^\phi\ (t{-}1)\ n) \textit{ else } X
\end{array}
$$

Defining trajectory of a formula is its defining sequence with the added constraints on state transitions imposed by the circuit model $\mathcal{M}$.

Definition 4. *Defining Trajectory*

$$
\begin{array}{lcl}
[\![f]\!]^\phi\ \mathcal{M}\ 0\ n & \triangleq & [f]^\phi\ 0\ n \\
[\![f]\!]^\phi\ \mathcal{M}\ t\ n & \triangleq & [f]^\phi\ t\ n \sqcup \mathcal{M}\ ([\![f]\!]^\phi\ \mathcal{M}\ (t{-}1))\ n
\end{array}
$$

Verification takes place by testing the validity of an *assertion* or property, of the form $(A \Rightarrow C)$, where both A and C are trajectory formulas. At the heart of STE model checking is an efficient implementation algorithm given by Theorem 1, that relies on the calculation of finite weakest sequences (defining sequence) and trajectories (defining trajectory) of the formulas, and comparing them (via the lattice ordering $\sqsubseteq$), point-wise for all nodes in C, up to the depth of the next-time operators in C.

Theorem 1. *STE implementation algorithm*

$$\mathcal{M} \models A \Rightarrow C \quad \triangleq \quad \forall t\, n.\ [C]^\phi\ t\ n \ \sqsubseteq\ [\![A]\!]^\phi\ \mathcal{M}\ t\ n$$

In the above theorem, the model is denoted by $\mathcal{M}$. The defining sequence is denoted by $[C]^{\phi}$ and defining trajectory by $[\![A]\!]^{\phi}$. Time is denoted by t, nodes by n and ϕ denotes the assignment of Boolean values to the variables that appear in the guard of the formula. Much of the debugging power of STE comes from the key observation that it is not necessary to supply ϕ in advance; instead the comparison is computed symbolically to give a constraint on ϕ, and represents precisely the conditions under which $A \Rightarrow C$ is true for the circuit.

B. Symmetry in circuit models and STE

We want to relate symmetries of the three-valued STE model to the STE property reduction. We are interested in the symmetries that arise due to permutations within the input and output groups of wires in each circuit. Every permutation can be defined in terms of a composition of swap operations. We therefore formalise the concept of permutation on states and sequences in terms of a swap operation on states and sequences respectively. Below, we provide the definition of these functions.

Definition 5. *Permutation on lattice states*

$$apply_s\,\pi\,s \;\triangleq\; \lambda\, n.\; s(\pi\; n)$$

Definition 6. *Permutation on lattice sequences*

$$apply_\sigma\,\pi\,\sigma \;\triangleq\; \lambda\, t\, n.\; \sigma\; t\; (\pi\; n)$$

Just as the swap function on node names can be composed serially to obtain arbitrary permutations on nodes, we compose the function $(apply_s\ \pi)$ and $(apply_\sigma\ \pi)$, with different underlying swaps given by π to obtain permutations on states and sequences respectively.

Every circuit model has some inputs that will contribute to the symmetry of the circuit, we call these symmetric inputs. Those inputs that do not have any role in the symmetry are called non-symmetric inputs. All these inputs and the output may contain several groups of wires, which in electrical terms is known as a bus. Symmetry generating permutations are applied to buses in the symmetric inputs and outputs. Since state is defined to be a function from node names to lattice values, applying permutations on the wires (node name) in a bus has an effect of permuting the state of the circuit.

One useful property of a swap function is that if the swap function exchanges a node $''a''$ with $''b''$ then applying the same swap on $''b''$ will give $''a''$. This is formalised below, by the predicate is_swap.

Definition 7. *Property of swap*

$$is\_swap\,\pi \;\triangleq\; \forall a\, b.\; (\pi\,(a) \;=\; b) \;\supset\; (\pi\,(b) \;=\; a)$$

We now present a formal definition of symmetry for STE models. The symbol χ denotes that the symmetry is for a three-valued logic containing Xs. Circuit models are denoted by $\mathcal{M}$, and s denotes the lattice valued states.

Definition 8. *Symmetry of STE models*

$$Sym_\chi\ \mathcal{M}\ \pi \;\triangleq\; \forall s.\; apply_s\,\pi\,(\mathcal{M}\,s) \;=\; \mathcal{M}\,(apply_s\,\pi\,s)$$

The above definition captures the intuition that if the behaviour of the circuit given by its model remains independent under permutation of its input and output states, then the circuit is said to have symmetry.

Generally, STE is all about evaluating sequences of states. Thus we need to know what happens to the information ordering preservation of two sequences if a permutation is applied on the sequences. The answer to this is in Lemma 1, which says that the information ordering is preserved under permutation of sequences.

Lemma 1. *Information ordering is preserved under permutation of sequences*

$$\begin{array}{l}\forall\pi.\; is\_swap\;\pi\;\supset\\ \quad\forall\sigma_1\,\sigma_2.\\ \qquad\forall t\,n.\;(\sigma_1\,t\,n \sqsubseteq \sigma_2\,t\;n\\ \qquad\qquad =\\ \qquad\qquad (apply_\sigma\;\pi\;\sigma_1)\;t\;n\;\sqsubseteq\;(apply_\sigma\,\pi\,\sigma_2)\;t\;n)\end{array}$$

Proof. The proof of this lemma follows from the definition of $apply_\sigma$, information ordering on sequences $\sqsubseteq$, and the fact that π is a swap. □

Now we define an application of permutation on a trajectory formula. The definition is by recursion on the structure of the formula.

Definition 9. *Permutation on formulas*

$$\begin{array}{lcl}apply_f\;\pi\;(n\;\mathsf{is}\;0) & \triangleq & (\pi\;n)\;\mathsf{is}\;0\\ apply_f\;\pi\;(n\;\mathsf{is}\;1) & \triangleq & (\pi\;n)\;\mathsf{is}\;1\\ apply_f\;\pi\;(f_1\;\mathsf{and}\;f_2) & \triangleq & (apply_f\;\pi\;f_1)\;\mathsf{and}\;(apply_f\;\pi\;f_2)\\ apply_f\;\pi\;(f\;\mathsf{when}\;G) & \triangleq & (apply_f\;\pi\;f)\;\mathsf{when}\;G\\ apply_f\;\pi\;(\mathsf{N}\;f) & \triangleq & \mathsf{N}\,(apply_f\;\pi\;f)\end{array}$$

Note that π is a swap on node names, and $(apply_f\ \pi)$ is composed for each underlying π to generate a complete permutation of any trajectory formula.

We now present an equivalence of applying permutations on a defining sequence of a formula and computing the defining sequence of the permutation of a formula. This is a vital instrument in establishing the concept, that permuting nodes in the formulas to generate new formulas will generate an equivalent defining sequence.

Lemma 2. *Permutation on the defining sequence and trajectory formulas*

$$\begin{array}{l}\forall\pi.\; is\_swap\;\pi\;\supset\\ \qquad\forall f\;\phi\;t\;n.\;(apply_\sigma\,\pi\,[f]^{\phi}\,t\,n\;=\;[apply_f\,\pi\,f]^{\phi}\,t\,n)\end{array}$$

We would like to prove a similar result about defining trajectory and trajectory formulas. Defining trajectories are sequences of states that the circuit evolves into, given a stimuli and the behaviour of the circuit model. Thus the role of circuit model becomes significant here. In order to ensure that the effect of permutation on the defining trajectory, is same as the effect of applying the permutation on the formula, and then computing the defining trajectory, we have to ensure that

the circuit model has symmetry. This is formally stated in the lemma below.

Lemma 3. *Permutation on the defining trajectory and trajectory formulas*

$$\begin{array}{l}\forall\pi.\ is\_swap\ \pi\ \supset\\ \quad\forall\mathcal{M}.\ Sym_{\chi}\ \mathcal{M}\ \pi\ \supset\\ \quad\quad\forall f\ \phi\ t\ n.\ (apply_{\sigma}\ \pi\ [\![f]\!]^{\phi}\ \mathcal{M}\ t\ n\ =\\ \quad\quad\quad\quad\quad\quad\quad\quad [\![apply_{f}\ \pi\ f]\!]^{\phi}\ \mathcal{M}\ t\ n)\end{array}$$

The theorem that encapsulates the relation between symmetric circuit models and the correctness results of symmetric STE properties is presented below.

Theorem 1. *Soundness Theorem*

$$\begin{array}{l}\forall\pi.\ is\_swap\ \pi\ \supset\\ \quad\forall\mathcal{M}.\ Sym_{\chi}\ \mathcal{M}\ \pi\ \supset\\ \quad\quad\forall A\,C.\ (\mathcal{M}\ \models\ A\ \Rightarrow\ C\ =\\ \quad\quad\quad\quad\quad\quad\quad\quad \mathcal{M}\ \models\ (apply_{f}\ \pi\ A)\ \Rightarrow\ (apply_{f}\ \pi\ C))\end{array}$$

This theorem is the most important result of this section. By using this theorem we explicitly verify one representative STE assertion from an equivalence class, and deduce the existence of the correctness of the entire class for symmetric circuits.

For proofs of the above lemmas and the soundness theorem, please refer to [11]. We have mechanised the proofs in the theorem prover HOL 4.

IV. STE INFERENCE RULES

In this section, we present the STE inference rules, and their proofs. Some of these rules, require the circuit model to be monotonic. Circuit models are represented as FSMs in Forte, and they are monotonic. We will use these inference rules to decompose the STE properties and re-use them in the reverse direction to re-compose smaller verification results.

The usage of STE inference rules for property decomposition is not new. Some of the rules that have been used in the past [7], [14] for decomposition include the rules on Reflexivity, Conjunction, Transitivity, Antecedent Strengthening 1 and Consequent Weakening 1. These are shown below. However, the rules we provide, not only help in decomposing the property into several smaller properties, but also expose the symmetric STE properties, to allow easy clustering.

Reflexivity

$$\frac{}{\mathcal{M}\models A\Rightarrow A}$$

Conjunction

$$\frac{\mathcal{M}\models A_1\Rightarrow B_1 \qquad \mathcal{M}\models A_2\Rightarrow B_2}{\mathcal{M}\models (A_1\ \mathsf{and}\ A_2)\Rightarrow(B_1\ \mathsf{and}\ B_2)}$$

Transitivity

$$\frac{\mathcal{M}\models A\Rightarrow B \qquad \mathcal{M}\models B\Rightarrow C}{\mathcal{M}\models A\Rightarrow C}$$

Antecedent Strengthening 1

$$\frac{\mathcal{M}\models A'\Rightarrow C \qquad [A']^{\phi}\sqsubseteq[A]^{\phi}}{\mathcal{M}\models A\Rightarrow C}$$

Consequent Weakening 1

$$\frac{\mathcal{M}\models A\Rightarrow C' \qquad [C]^{\phi}\sqsubseteq[C']^{\phi}}{\mathcal{M}\models A\Rightarrow C}$$

We have extended the language of the STE inference rules by adding a layer of propositional logic on top of the language of the STE. This is done by providing a rule that enables us to transform a property with the guards in the consequent of a property, to another equivalent property where the guards can be pushed out as an implication. This is given by the rule known as Constraint Implication 1 shown below.

The reason for doing this, is an observation that many STE properties we encounter during the verification of symmetric circuits, can be effectively decomposed, by targeting the decomposition of the Boolean guards and trajectory formulas. The inference system looks at the structure of both the Boolean expressions, and the trajectory formulas, and works simultaneously to reduce both of them. It is done easily when the guards are pushed out of the trajectory formula. The decomposition of guards and the formulas in this manner has not been exploited in the past with STE model checking.

Constraint Implication 1

$$\frac{\mathcal{M}\ \models\ A\Rightarrow(C\ \mathsf{when}\ G)}{G\supset(\mathcal{M}\ \models\ A\Rightarrow C)}$$

Proof: The proof takes place by a case split on the Boolean guard G. When G is true, the premise of the rule reduces to $\mathcal{M}\models(A\Rightarrow C)$, from the definition of the STE implementation algorithm. Thus the conclusion of the rule holds. When G is false the rule holds immediately. □

The other direction is the one in which the assumption of a Boolean implication can be pushed inside the syntax of the STE property. This is given by the rule Constraint Implication 2 (Constr Impl 2).

Constraint Implication 2

$$\frac{G\supset(\mathcal{M}\ \models\ A\Rightarrow C)}{\mathcal{M}\ \models\ A\Rightarrow(C\ \mathsf{when}\ G)}$$

Proof: By doing a case split on the Boolean guard G, we discharge the case when G is true, by using the definition of the STE implementation algorithm, and the case when G is false, follows by using the definition of the STE implementation, and observing that $\forall a.X\sqsubseteq a$. □

The following rule allows a property with a conjunctive Boolean assumption to be decomposed into smaller properties with the smaller individual conjuncts. This rule called Guard Conjunction (Grd Conj) turns out to be very useful for proving STE properties as we will show when we present the examples.

Guard Conjunction

$$\frac{G_1 \supset (\mathcal{M} \models A \Rightarrow C) \qquad G_2 \supset (\mathcal{M} \models B \Rightarrow D)}{(G_1 \wedge G_2) \supset (\mathcal{M} \models (A \text{ and } B) \Rightarrow (C \text{ and } D))}$$

Proof: The proof follows easily from the assumptions, and the Conjunction rule of inference. □

The next rule appears to be the dual of the Guard Conjunction rule, but it is more subtle. This is because the assumption in the consequent of the rule is a disjunction, and the consequent of the STE property has only one formula C. What this rule attempts to capture is that if we can verify two STE properties against a given circuit model, with separate antecedents and separate Boolean assumptions, but identical consequent, then we can do a disjunction of the Boolean assumptions and the conjunction of the antecedents of the two STE properties to deduce that the resulting STE property is satisfied by the circuit model. This rule known as the Guard Disjunction (Grd Disj), becomes very useful in decomposing properties that have a Boolean disjunction in the guard of the STE formula. These are also some of the more notoriously difficult properties to verify, because of the disjunction in the Boolean guard.

Guard Disjunction

$$\frac{G_1 \supset (\mathcal{M} \models A \Rightarrow C) \qquad G_2 \supset (\mathcal{M} \models B \Rightarrow C)}{G_1 \vee G_2 \supset (\mathcal{M} \models (A \text{ and } B) \Rightarrow C)}$$

Proof: Assuming G_1 holds, the conclusion of the rule follows from the premise, and using the fact that for monotonic circuit models, $\forall \mathcal{M} A B.(\forall t\, n.[\![A]\!]^{\phi}\, \mathcal{M}\, t\, n \;\sqsubseteq\; [\![A \text{ and } B]\!]^{\phi}\, \mathcal{M}\, t\, n)$. Assuming G_2 holds, the conclusion follows again, using the same argument as in the case when G_1 was true. □

The last rule we present here is known as the Cut. This rule enables the verification of a given STE property with conjunctive Boolean guards to be done by decomposing the property verification into two parts, in a way that we verify an intermediate property using an intermediate formula, and a part of the Boolean guards of the original STE property, then use this intermediate formula together with the other half of the original antecedent, to verify the property with the consequent of the original STE property, under the presence of the other Boolean conjunct.

Cut

$$\frac{G_1 \supset (\mathcal{M} \models A_1 \Rightarrow B_1) \qquad G_2 \supset (\mathcal{M} \models (B_1 \text{ and } A_2) \Rightarrow C)}{(G_1 \wedge G_2) \;\supset\; (\mathcal{M} \models (A_1 \text{ and } A_2) \Rightarrow C)}$$

Proof: From the Conjunction rule of inference, we get: $\mathcal{M} \models (A_1 \text{ and } B_1) \supset (\mathcal{M} \models A_2 \Rightarrow A_2) \supset (\mathcal{M} \models (A_1 \text{ and } A_2) \Rightarrow (B_1 \text{ and } A_2))$. From the Reflexivity rule of inference we know that $\mathcal{M} \models A_2 \Rightarrow A_2$. Together with the premise, using the modus ponen, we get $\mathcal{M} \models (A_1 \text{ and } A_2) \Rightarrow (B_1 \text{ and } A_2)$. Using the Transitivity rule, on this and the assumptions, we obtain the conclusion. □

Antecedent Strengthening 2

$$\frac{G \supset (\mathcal{M} \models A' \Rightarrow C) \qquad [A']^{\phi} \sqsubseteq [A]^{\phi}}{G \supset (\mathcal{M} \models A \Rightarrow C)}$$

Proof: The proof follows easily from Antecedent Strengthening 1 and the assumptions. □

Consequent Weakening 2

$$\frac{G \supset (\mathcal{M} \models A \Rightarrow C') \qquad [C]^{\phi} \sqsubseteq [C']^{\phi}}{G \supset (\mathcal{M} \models A \Rightarrow C)}$$

Proof: The proof follows easily from Consequent Weakening 1, and the assumptions. □

We have implemented the proofs of all the inference rules in HOL.

V. A SIMPLE EXAMPLE — UNIT-DELAY COMPARATOR

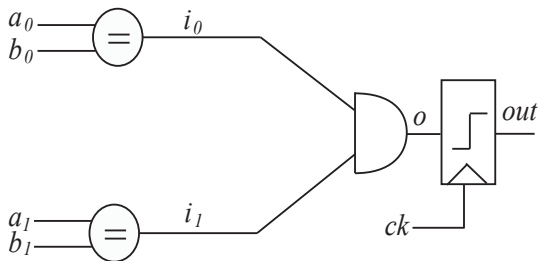


Fig. 2. A unit-delay 2-bit comparator

The comparator circuit in Figure 2, has symmetry across its two input buses and the output. We can swap $''a_0''$ with $''a_1''$ and $''b_0''$ with $''b_1''$ and the output with itself, and the behaviour of the circuit will not change. This is the property that we will exploit in the verification of the STE property.

The property we wish to verify for the comparator is given by the following:

$$\begin{aligned} Comp \;\models\; & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\ & (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\ & \Rightarrow \mathsf{N}((''out'' \text{ is } 1) \text{ when} \\ & \qquad ((a_0 = b_0) \wedge (a_1 = b_1))) \text{ and} \\ & \mathsf{N}((''out'' \text{ is } 0) \text{ when} \\ & \qquad (\neg(a_0 = b_0) \vee (\neg(a_1 = b_1)))) \end{aligned}$$

The above property can be decomposed using the Conjunction rule into the following two cases:

I. Equality case

$$\begin{aligned} Comp \;\models\; & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\ & (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\ & \Rightarrow \mathsf{N}((''out'' \text{ is } 1) \text{ when} \\ & \qquad ((a_0 = b_0) \wedge (a_1 = b_1))) \end{aligned}$$

II. Inequality case

$$\begin{aligned} Comp \;\models\; & (''a_0'' \text{ is } a_0) \text{ and } (''b_0'' \text{ is } b_0) \text{ and} \\ & (''a_1'' \text{ is } a_1) \text{ and } (''b_1'' \text{ is } b_1) \\ & \Rightarrow \mathsf{N}((''out'' \text{ is } 0) \text{ when} \\ & \qquad (\neg(a_0 = b_0) \vee (\neg(a_1 = b_1)))) \end{aligned}$$

We will show how to verify the equality case first. The verification strategy relies on the inference rules. We explain the steps below. We will use the Cut rule to partition the verification of a property into two cases, which rely on an intermediate trajectory formula. Below these intermediate formulas are given by B_0 and B_1. Other trajectory formulas describing parts of the antecedent and the consequent of the given property are defined.

$$\begin{array}{lcl} let\ B_0 & = & (''i_0''\ \mathsf{is}\ 1) \\ let\ B_1 & = & (''i_1''\ \mathsf{is}\ 1) \\ let\ A_0 & = & (''a_0''\ \mathsf{is}\ a_0)\ \mathsf{and}\ (''b_0''\ \mathsf{is}\ b_0) \\ let\ A_1 & = & (''a_1''\ \mathsf{is}\ a_1)\ \mathsf{and}\ (''b_1''\ \mathsf{is}\ b_1) \\ let\ G_0 & = & (a_0 = b_0) \\ let\ G_1 & = & (a_1 = b_1) \\ let\ C' & = & (''o''\ \mathsf{is}\ 1) \\ let\ C & = & \mathsf{N}(''out''\ \mathsf{is}\ 1) \end{array}$$

Here is how the verification will proceed once the trajectory formulas have been defined. It begins by carrying out an STE run verifying the weaker property i.e., the result of bitwise comparison of the values at nodes $''a_0''$ and $''b_0''$ is 1, if the values are equal. This run requires two symbolic variables a_0 and b_0. By symmetry, it can be concluded by using Theorem 1 that the other part of the bitwise comparison circuitry is also correct. Then Constraint Implication rules transform the properties such that the guards are moved out of the consequent of the properties and a Guard Conjunction rule is used to stitch the correctness results of the two bitwise comparison cases. Then assuming that all inputs to the And gate of the comparator are 1, we check if the output is 1. This run basically uses the intermediate formulas in the antecedent and carries out the scalar simulation in Forte. Another straightforward scalar run is carried out to check if the output value appears at $''out''$ one time point later than $''o''$. Finally by virtue of Transitivity rule and the Cut, we deduce the correctness property we started out to do. The property is finally transformed using the Constraint Implication 2 rule, to transform it to exactly the property we wanted to verify in the first instance. All the steps are outlined below:

1.	$Comp \models A_0 \Rightarrow (B_0\ \mathsf{when}\ G_0)$	(*STE run*)
2.	$Comp \models A_1 \Rightarrow (B_1\ \mathsf{when}\ G_1)$	(*Symmetry*)
3.	$G_0 \supset (Comp \models A_0 \Rightarrow B_0)$	(*Constr Impl 1*)
4.	$G_1 \supset (Comp \models A_1 \Rightarrow B_1)$	(*Constr Impl 1*)
5.	$(G_0 \wedge G_1) \supset (Comp \models (A_0\ \mathsf{and}\ A_1) \Rightarrow (B_0\ \mathsf{and}\ B_1))$	(*Grd Conj*)
6.	$Comp \models (B_0\ \mathsf{and}\ B_1) \Rightarrow C'$	(*STE run*)
7.	$Comp \models C' \Rightarrow C$	(*STE run*)
8.	$Comp \models (B_0\ \mathsf{and}\ B_1) \Rightarrow C$	(*Trans on 6 and 7*)
9.	$(G_0 \wedge G_1) \supset (Comp \models (A_0\ \mathsf{and}\ A_1) \Rightarrow C)$	(*Cut on 5 and 8*)
10.	$Comp \models (A_0\ \mathsf{and}\ A_1) \Rightarrow (C\ \mathsf{when}\ (G_0 \wedge G_1))$	(*Constr Impl 2*)

Replacing the values of A_0, A_1, C, G_0 and G_1 we can see that we have verified the equality case. Now we will show how to verify the inequality case. The formulas A_0 and A_1 remain the same, but we redefine C', C, and the guards G_0 and G_1.

$$\begin{array}{lcl} let\ C' & = & (''o''\ \mathsf{is}\ 0) \\ let\ C & = & \mathsf{N}(''out''\ \mathsf{is}\ 0) \\ let\ G_0 & = & \neg(a_0 = b_0) \\ let\ G_1 & = & \neg(a_1 = b_1) \end{array}$$

Now we proceed as follows:

1.	$Comp \models A_0 \Rightarrow (C'\ \mathsf{when}\ G_0)$	(*STE run*)
2.	$Comp \models A_1 \Rightarrow (C'\ \mathsf{when}\ G_1)$	(*Symmetry*)
3.	$G_0 \supset (Comp \models A_0 \Rightarrow C')$	(*Constr Impl 1*)
4.	$G_1 \supset (Comp \models A_1 \Rightarrow C')$	(*Constr Impl 1*)
5.	$(G_0 \vee G_1) \supset (Comp \models ((A_0\ \mathsf{and}\ A_1) \Rightarrow C'))$	(*Grd Disj 3, 4*)
6.	$Comp \models C' \Rightarrow C$	(*STE run*)
7.	$(G_0 \vee G_1) \supset (Comp \models ((A_0\ \mathsf{and}\ A_1) \Rightarrow C))$	(*Cut*)

Thus the number of variables required in any STE run is 2. By using inference rules we are able to decompose the properties effectively, and using symmetry collapses the number of explicit cases to verify to just one, and that case needs 2 variables. Thus variables required for verifying an n-bit comparator remains fixed at 2 and not $2n$. So BDDs always stay small and verification complexity remains constant with respect to BDD size.

VI. A CIRCUIT WITH MULTIPLE CAMS — CASE STUDY

Our final case study is that of a circuit that contains two CAMs. We call the circuit model cam_xor. The property we wish to verify about this circuit is that the incoming tag can only be found in at most one CAM. If a tag is found in more than one CAM, hit (out) stays low. Figure 3 shows the circuit. Circuits with multiple CAMs are abundantly found in caches. The associativity in a cache determines the number of CAMs used in the cache, and the number of comparators used in parallel for comparing the incoming tag address with the stored tags in different CAMs. The principle we illustrate in the verification of the circuit in this section, can be applied to the verification of a cache.

For the sake of simplicity in illustration, we consider the case with two lines in each CAM, and the tag width being two bits, and we leave out the associated data part of the CAMs. This is sufficient to show the principle involved in the verification using symmetry. Any larger circuit can be verified using exactly the same principle. There is symmetry amongst tag and data bits in the CAM. The symmetry in tag bits is useful for collapsing the set of verification runs for tag comparison properties. This is because comparators that have symmetry, are used in this circuit, and therefore we re-use the correctness results from the comparator verification case here. The symmetry in data bits is useful for collapsing the case of verifying correct data read/write property. However, due to the lack of space we do not show the verification of that property here. The interested reader is encouraged to see [5].

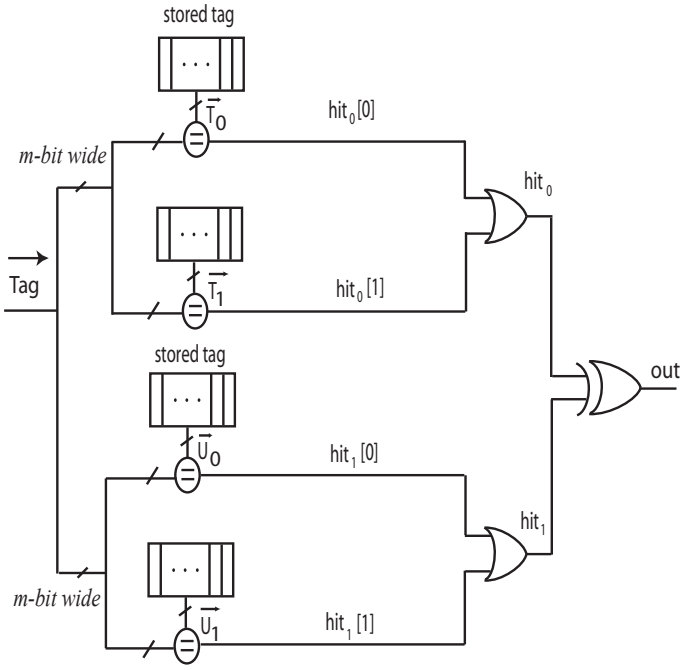


Fig. 3. A circuit with two CAMs, showing the tag comparison circuitry, the incoming tags match at most one line in the CAM, and hit rises only if the tag is found in at most one of the CAMs but not both. We consider the case of two lines in each CAM, with two bit wide tags in this example. Stored tags for the first CAM are denoted by T, and for the second CAM by the letter U. The subscript (0 or 1) denotes which line in the CAM (1st or 2nd) the tags are stored. Circuits like these, where an incoming tag is compared in parallel with tags in multiple CAMs appear in all modern day set-associative caches.

We assume the existence of symbolic (BDD) variables for the tag and data lines. We state the basic antecedent assertion that assigns the symbolic incoming tag values to the incoming tag nodes. When the *tagen* signal is low, new tags are written into the CAM, and when the *tagen* signal is high, tags can be read out from the CAM lines. For the sake of simplicity of presentation here, we have left out the tag enable signal from the figure.

// tag enabled and incoming tags take symbolic values
$let\ base\_ant\ =\ ((((''Tag[0]''\ \text{is}\ Tag_0)$
$\text{and}\ (''Tag[1]''\ \text{is}\ Tag_1))\ from\ 0\ to\ 2)$
$\text{and}\ (''tagen''\ \text{is}\ F\ from\ 0\ to\ 1)\ \text{and}$
$(''tagen''\ \text{is}\ T\ from\ 1\ to\ 2)$

We then provide the trajectory formulas stating that the first and second lines in each CAM are populated with symbolic tag values.

// populate the 1st line of the 1st CAM
$let\ A_0\ =\ ((((''T_0[0]''\ \text{is}\ t_{00})\ \text{and}$
$(''T_0[1]''\ \text{is}\ t_{01}))\ from\ 0\ to\ 2)\ \text{and}\ base\_ant$

// populate the 2nd line of the 1st CAM
$let\ B_0\ =\ ((((''T_1[0]''\ \text{is}\ t_{10})\ \text{and}$
$(''T_1[1]''\ \text{is}\ t_{11}))\ from\ 0\ to\ 2)\ \text{and}\ base\_ant$

Similarly, the first and the second lines of the second CAM are populated with symbolic values.

// populate the 1st line of the 2nd CAM
$let\ A_1\ =\ ((((''U_0[0]''\ \text{is}\ u_{00})\ \text{and}$
$(''U_0[1]''\ \text{is}\ u_{01}))\ from\ 0\ to\ 2)\ \text{and}\ base\_ant$

// populate the 2nd line of the 2nd CAM
$let\ B_1\ =\ ((((''U_1[0]''\ \text{is}\ u_{10})\ \text{and}$
$(''U_1[1]''\ \text{is}\ u_{11}))\ from\ 0\ to\ 2)\ \text{and}\ base\_ant$

We then define the trajectory formulas that state the desired consequent of the property, that the hit was found in the first and the second CAMs.

// hit found in the 1st CAM
$let\ C_0\ =\ ''hit_0''\ \text{is}\ T\ from\ 1\ to\ 2$

// hit found in the 2nd CAM
$let\ C_1\ =\ ''hit_1''\ \text{is}\ T\ from\ 1\ to\ 2$

Similarly we can define the formulas that state that the hit was not found in the first and the second CAM.

// hit not found in the 1st CAM
$let\ D_0\ =\ ''hit_0''\ \text{is}\ F\ from\ 1\ to\ 2$

// hit not found in the 2nd CAM
$let\ D_1\ =\ ''hit_1''\ \text{is}\ F\ from\ 1\ to\ 2$

We also provide the formulas that state where in each CAM the hit was found. To save space, we only show the formulas for the first CAM.

// hit not found at 1st line of 1st CAM
$let\ D_{00}\ =\ ''hit_0[0]''\ \text{is}\ F\ from\ 1\ to\ 2$

// hit not found at 2nd line of 1st CAM
$let\ D_{01}\ =\ ''hit_0[1]''\ \text{is}\ F\ from\ 1\ to\ 2$

The following Boolean formulas state where the incoming tags match the stored tags of the CAM lines. We use the letter G to denote the Boolean formulas that state that the tags match at the lines of the first CAM, while we use the letter H to denote the Boolean formula that states that tags match at the lines of the second CAM. The suffixes distinguish the line in each CAM, where the tags are found (0 for the first CAM and 1 for the second CAM).

// tags match at 1st line of 1st CAM
$let\ G_0\ =\ (Tag_0\ =\ t_{00})\ \wedge\ (Tag_1\ =\ t_{01})$

// tags match at 2nd line of 1st CAM
$let\ G_1\ =\ (Tag_0\ =\ t_{10})\ \wedge\ (Tag_1\ =\ t_{11})$

// tags match at 1st line of 2nd CAM
$let\ H_0\ =\ (Tag_0\ =\ u_{00})\ \wedge\ (Tag_1\ =\ u_{01})$

// tags match at 2nd line of 2nd CAM
$let\ H_1\ =\ (Tag_0\ =\ u_{10})\ \wedge\ (Tag_1\ =\ u_{11})$

0-7695-2707-8/06 $20.00

The verification of the property involves using some verification results for a CAM. The property we are looking to verify for a CAM is that if the incoming tags match at any line in the CAM then the hit from the CAM rises, and if the tags don't match at any line, then the hit stays low. We assume the invariant that the tags match at most one line in the CAM. We will show the steps involved in verifying this property below.

A. Verifying the hit circuitry of the CAM

One of the desired properties we wish to verify for a CAM, is that if the incoming tags match at any of the lines in the CAM, then a hit signal should be flagged at the output of the CAM. This is done by checking the value of the hit output in the CAM. It is flagged as high if the match was found, else it stays low. We show the verification steps for this property for the 1st CAM, and the same steps are used for the verification of the property for the 2nd CAM, so shall not be shown here. Since the tag comparison circuitry uses the comparator circuit we showed earlier, we can re-use the verification results for the comparator here. We refer to the verification strategy used for the comparator verification earlier as the *Comp result*, in the sequel below. We will first show the verification of the property that states that the hit rises if there is a match in the first CAM. This is shown in Steps 1-3 below.

1. $G_0 \supset (cam\_xor \models (A_0 \Rightarrow C_0)$ (*Comp result*)
2. $G_1 \supset (cam\_xor \models (B_0 \Rightarrow C_0)$ (*Comp result*)
3. $(G_0 \vee G_1) \supset$
   $(cam\_xor \models (A_0 \text{ and } B_0) \Rightarrow C_0)$ (*Grd Disj*)

If the incoming tags don't match at any line in the first CAM, then the hit stays low. The verification of this property is shown in the following steps. We use the correctness result of the comparator for inequality case, and do a Guard Conjunction to obtain Step 4.

4. $(\neg G_0 \wedge \neg G_1) \supset$
   $(cam\_xor \models (A_0 \text{ and } B_0) \Rightarrow (D_{00} \text{ and } D_{01}))$

Then a scalar run is done to verify the property that if the hit from both the lines from the first CAM is low, then the hit from the first CAM stays low.

5. $cam\_xor \models (D_{00} \text{ and } D_{01}) \Rightarrow D_0$

Finally we do a Cut on Step 4 and 5 to obtain the correctness of the property that says that if the match was not found in any line of the first CAM then the hit from that CAM stays low.

6. $(\neg G_0 \wedge \neg G_1) \supset$
   $(cam\_xor \models (A_0 \text{ and } B_0) \Rightarrow D_0)$

The property verification for the other CAM can be done by repeating the above steps. Once we have verified the properties for both the CAMs, we are now ready to verify the property that the hit can at most be found in one of the CAMs. We show the steps involved in verifying this property using the verification results for individual CAMs shown above, and using STE runs and inference rules. We shall refer to the verification results for the individual CAM as the *CAM result*, when we show the steps below.

7. $(H_0 \vee H_1) \supset$
   $(cam\_xor \models (A_1 \text{ and } B_1) \Rightarrow C_1)$ (*CAM result*)
8. $(\neg H_0 \wedge \neg H_1) \supset$
   $(cam\_xor \models (A_1 \text{ and } B_1) \Rightarrow D_1)$ (*CAM result*)
9. $cam\_xor \models (C_0 \text{ and } C_1)$
   $\Rightarrow (''out'' \text{ is } F)$ (*scalar run*)
10. $(G_0 \vee G_1) \wedge (H_0 \vee H_1) \supset$
    $(cam\_xor \models (A_0 \text{ and } A_1) \text{ and }$
    $(B_0 \text{ and } B_1)$
    $\Rightarrow (C_0 \text{ and } C_1))$ (*Grd Conj 3, 7*)
11. $(G_0 \vee G_1) \wedge (H_0 \vee H_1) \supset$
    $(cam\_xor \models (A_0 \text{ and } A_1) \text{ and }$
    $(B_0 \text{ and } B_1)$
    $\Rightarrow (''out'' \text{ is } F)$ (*Cut 9, 10*)
12. $((\neg G_0 \wedge \neg G_1) \wedge (H_0 \vee H_1)) \supset$
    $(cam\_xor \models (A_0 \text{ and } A_1) \text{ and }$
    $(B_0 \text{ and } B_1)$
    $\Rightarrow (D_0 \text{ and } C_1))$ (*Grd Conj 6, 7*)
13. $((\neg H_0 \wedge \neg H_1) \wedge (G_0 \vee G_1)) \supset$
    $(cam\_xor \models (A_0 \text{ and } A_1) \text{ and }$
    $(B_0 \text{ and } B_1)$
    $\Rightarrow (D_1 \text{ and } C_0))$ (*Grd Conj 3, 8*)
14. $cam\_xor \models (D_0 \text{ and } C_1) \Rightarrow$
    $(''out'' \text{ is } T)$ (*scalar run*)
15. $cam\_xor \models (D_1 \text{ and } C_0) \Rightarrow$
    $(''out'' \text{ is } T)$ (*scalar run*)
16. $((\neg G_0 \wedge \neg G_1) \wedge (H_0 \vee H_1)) \supset$
    $(cam\_xor \models (A_0 \text{ and } A_1) \text{ and }$
    $(B_0 \text{ and } B_1)$
    $\Rightarrow (''out'' \text{ is } T))$ (*Cut 12, 14*)
17. $((\neg H_0 \wedge \neg H_1) \wedge (G_0 \vee G_1)) \supset$
    $(cam\_xor \models (A_0 \text{ and } A_1) \text{ and }$
    $(B_0 \text{ and } B_1)$
    $\Rightarrow (''out'' \text{ is } T))$ (*Cut 13, 15*)

Property (11), (16) and (17) are the properties we had set out to verify, and we verify these using only two BDD variables. This is because we need only two variables for verifying an $m-bit$ comparator, and since symmetry in the circuit with multiple CAMs is due to the symmetry in comparators, one condenses the requirement of variables for explicit verification runs for this circuit, to the requirement for the explicit verification runs for the comparator circuit.

VII. Related work

Several researchers in the past [2], [3], [6], [9], [11] have investigated the usage of symmetry for performing reductions for model checking. The work that comes closest to ours is Pandey's [11], [12]. In [11], he used symmetry reduction for verifying SRAMs. The exact number of variables required in the verification was not reported, however verification memory used was shown. Comparing that to the number of variables we used in verifying SRAMs, we perform slightly better (for details see [5]). This is due to the fact that we use

symmetry and symbolic indexing both, whereas Pandey has used only symmetries for SRAM verification. For CAMs, Pandey et al. [12] used symbolic indexing, and reported logarithmic reduction in the number of variables required. By using symmetries for CAMs and CAM like circuits like the one we showed in this paper, we only need a fixed small number of BDD variables. Our approach relies on using property decomposition rules where the amount of time used in verification is linear with respect to the tag width, number of CAM lines and the number of CAMs. A key distinguishing factor between Pandey's work and our's is the way symmetries are discovered in the circuit models. Whereas Pandey had used NP-complete sub-graph isomorphism based techniques to discover symmetries in transistor level netlist representation of models, we capture symmetries in the structure of the model at the time of design, and later infer their presence by type checking [4], [5]. The idea of using special types to record symmetries in systems was advocated by Ip and Dill [3]. They used Murphi, a guarded command language with scalarsets, to model their system and used symmetry reduction for verification of several cache-coherence protocols. McMillan also used type based inference for symmetry detection [9]. He used refinement relations to decompose the verification problem into smaller runs, and subsequently used symmetry reduction together with other reduction techniques such as path splitting and data type reduction. Clarke et al. [2] investigated the idea of using symmetries for reduction of state spaces in model checking. Their solution is applicable for both the cases, when the transition relation is provided explicitly in terms of states, and when it is given symbolically as a BDD. They show that for CTL* and Mu-calculus, model checking of the original model can be reduced to the problem of model checking a smaller model which is a quotient of the original one. The quotient is obtained by taking an intersection of the symmetries of the model and the properties. This idea was also used by Emerson and Sistla [6]. who describe a method of exploiting symmetry based reduction for CTL* model checking. Symmetry in models was read directly from the program source by monitoring the process indices, whereas symmetry amongst properties was identified by manual inspection. In a recent work, Sistla and Godefroid [15] provide further extensions to this approach, by using guarded annotated quotient structures thus enabling symmetries to be identified for states that are variable-value pairs besides process indices.

VIII. Conclusion and Future Work

We have presented a reduction methodology for STE model checking that exploits the symmetry in circuit and uses a novel set of inference rules for property decomposition, in a way that the BDD variable requirement is reduced to a constant number variables for CAMs and CAM like circuits. The rules are generic, and they can be used in the verification of non-symmetric systems as well. Using inference rules effectively requires manual inspection at the moment. In the future, we would like to use these rules to be used for automatic property decomposition in a manner that reveals the symmetry amongst properties. We showed that the new set of inference rules is sound, however we did not discuss the completeness issues, future work should look into that.

IX. Acknowledgement

Thanks to my supervisor Tom Melham for providing an opportunity to work on this problem, and giving useful feedback. Urban Ingelsson, Susanto Kong and Ed Smith gave useful comments on the paper. This work benefited in its early stages from discussions with John O'Leary and Robert Jones. This research was funded in part by a donation from Intel Corp. USA, and Overseas Research Studentship, UK.

References

[1] M. D. Aagaard and C.-J. H. Seger, "The Formal Verification of a Pipelined Double-Precision IEEE Floating-Point Multiplier," in *International Conference on Computer-Aided Design*. IEEE Computer Society, nov 1995, pp. 7–10.

[2] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn, "Exploiting Symmetry in Temporal Logic Model Checking," *Formal Methods in System Design: An International Journal*, vol. 9, no. 1/2, pp. 77–104, August 1996.

[3] C.N. Ip and D.L. Dill, "Better verification through symmetry," in *Computer Hardware Description Languages and their Applications*, D. Agnew, L. Claesen, and R. Camposano, Eds. Ottawa, Canada: Elsevier Science Publishers B.V., Amsterdam, Netherland, 1993, pp. 87–100.

[4] A. Darbari, "A Structured Modelling Framework to capture Symmetry in Ciruits," in *Proceedings of the 13th Workshop on Automated Reasoning*, University of Bristol, Bristol, April 2006.

[5] A. Darbari, "Symmetry Reduction for STE Model Checking using Structured Models," Ph.D. dissertation, Oxford University Computing Lab, Wolfson Building, Parks Road, Oxford, Submitted 2006.

[6] F. A. Emerson and A. P. Sistla, "Symmetry and Model Checking," *Journal of Formal Methods in System Design*, vol. 9, no. 1/2, pp. 105–131, August 1996. [Online]. Available: citeseer.ist.psu.edu/emerson94symmetry.html

[7] S. Hazelhurst and C.-J. H. Seger, "A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDDs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, vol. 14, no. 4, pp. 413–422, April 1995.

[8] R. B. Jones, J. W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical Formal Verification in Microprocessor Design," *IEEE Design & Test of Computers*, vol. 18, no. 4, pp. 16–25, July/August 2001.

[9] K L. McMillan, "A Methodology for Hardware Verification using Compositional Model Checking," *Science of Computer Programming*, vol. 37, no. 1-3, pp. 279–309, 2000. [Online]. Available: citeseer.nj.nec.com/mcmillan99methodology.html

[10] T. Melham and A. Darbari, "Symbolic Trajectory Evaluation in a Nutshell," unpublished report, available on request.

[11] M. Pandey, "Formal Verification of Memory Arrays," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1997.

[12] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir, "Formal Verification of Content Addressable Memories Using Symbolic Trajectory Evaluation," in *DAC '97: Proceedings of the 34th annual conference on Design automation*. ACM Press, New York, NY, USA, 1997, pp. 167–172.

[13] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Journal of Formal Methods in System Design*, vol. 6, no. 2, pp. 147–189, March 1995.

[14] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 9, pp. 1381–1405, September 2005. [Online]. Available: http://www.comlab.ox.ac.uk/tom.melham/pub/Seger-2005-IEE.pdf

[15] A. P. Sistla and P. Godefroid, "Symmetry and reduced symmetry in model checking." *ACM Transactions on Programming Languages and Systems*, vol. 26, no. 4, pp. 702–734, 2004.

Thorough Checking Revisited

Shiva Nejati, Mihaela Gheorghiu, and Marsha Chechik
Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
Email:{shiva,mg,chechik}@cs.toronto.edu

***Abstract*— Recent years have seen a proliferation of 3-valued models for capturing abstractions of systems, since these enable verifying both universal and existential properties. Reasoning about such systems is either inexpensive and imprecise (compositional checking), or expensive and precise (thorough checking). In this paper, we prove that thorough and compositional checks for temporal formulas in their disjunctive forms coincide, which leads to an effective procedure for thorough checking of a variety of abstract models and the entire μ-calculus.**

I. Introduction

Recent years have seen a proliferation of approaches to capturing abstract models using rich formalisms that enable reasoning about arbitrary temporal properties. Examples of such formalisms include *Partial Kripke Structures (PKSs)* [1], *Mixed Transition Systems (MixTSs)* [2], [3], *Hyper-Transition Systems (HTSs)* [4], [5], [6], etc. Model checking over these is either conclusive, i.e., the property of interest can be proven or refuted, or inconclusive, denoted maybe, indicating that the abstract model needs to be refined.

Two distinct 3-valued semantics of temporal logic are used over these abstract models. One is *compositional*, in which the value of a property is computed from the values of its subproperties (as in classical model checking), and the other one is *thorough* [1]. The latter assigns maybe to a property only if there is a pair of concretizations of the abstract model such that the property holds in one and fails in the other. In general, model checking with thorough semantics is more expensive than compositional model checking – EXPTIME-complete for CTL, LTL and μ-calculus (L_μ) [7]. Thorough semantics, however, is more conclusive than compositional. For example, consider the program P shown in Figure 1(b), where `x` and `y` are integer variables and `x, y = e1, e2` indicates that `x` and `y` are simultaneously assigned `e1` and `e2`, respectively. A PKS M, shown in Figure 1(c), is an abstraction of P w.r.t. predicates p (meaning "`x` is odd"), and q, (meaning "`y` is odd"). The CTL formula $\varphi = AGq \wedge A[pU\neg q]$ evaluates to maybe on M under compositional semantics and to false under thorough, since every refinement of M refutes φ.

For the purpose of effective reasoning about abstract models, it is important to enable thorough-quality analysis using (compositional) 3-valued model checking. Specifically, we aim to identify classes of temporal formulas whose compositional model checking is as precise as thorough. Otherwise, we want to transform the formulas into equivalent ones (in classical logic), for which compositional checking yields the most precise answer. For example, we would transform the formula $AGq \wedge A[pU\neg q]$ into $A[p \wedge q\ U\ \mathsf{false}]$, that is unsatisfiable (over total models) and thus always false. [9], [8] refer to this process as *semantic minimization*, and the formulas for which thorough and compositional semantics coincide as *self-minimizing*.

This paper addresses thorough checking of L_μ formulas over various abstract models with 3-valued semantics following the algorithm in Figure 1(a). This algorithm consists of three main steps: (1) (compositional) model checking of φ over an abstract model M (e.g.,[1], [2], [5]), (2) checking if φ is self-minimizing, and (3) computing semantic minimization of φ, and then model checking the resulting formula. Computing semantic minimization is the most expensive part and is at least as hard as thorough checking [8]. Therefore, it is important to identify as many self-minimizing formulas as possible in step (2), and avoid step (3).

In [8] and [10], it was shown that positive/monotone temporal formulas, i.e., the ones that do not include both p and $\neg p$, are self-minimizing over abstract models described as PKSs. This self-minimization check, however, is not robust for more expressive abstraction modelling formalisms such as HTSs. For example, consider an abstraction of program P, described as an HTS H, shown in Figure 1(d). Based on the 3-valued semantics of L_μ over HTSs, the monotone formula $AGp \wedge AGq$ evaluates to maybe over H [5], [4]. However, this formula is false by thorough checking, because every concretization of H refutes either AGp or AGq.

In this paper, we extend step (2) of the thorough checking algorithm by proving that the disjunctive and conjunctive normal forms of L_μ defined in [11] are self-minimizing over abstract models described as HTSs. We focus on HTSs because other 3-valued abstraction formalisms can be translated to them without loss of precision, but not the other way around [4]. Godefroid and Huth [8] proved that *monotone* disjunctive L_μ formulas *without* greatest fixpoints are self-minimizing for PKSs, and pointed out that by a naive inductive proof the self-minimization of greatest fixpoint disjunctive formulas cannot be shown. We improve on this result by using an automata intersection game inspired by [12], to show that the disjunctive and conjunctive normal forms of L_μ *with* greatest fixpoint are self-minimizing over HTSs. Our result yields a simple syntactic check for identifying self-minimizing formulas over HTSs and can be used along with the monotonicity condition for PKSs and MixTSs.

Our result further provides an alternative semantic minimization procedure for step (3) of the algorithm, via the

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

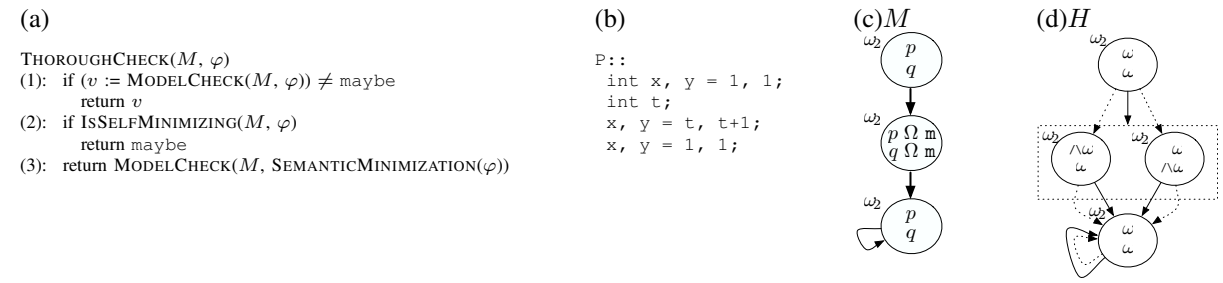


Fig. 1. (a) A sketch of an algorithm for thorough checking. A simple program P (adapted from [8]) (b) and its abstractions described as: (c) a PKS M; and (d) an HTS H.

tableau-based translation of Janin and Walukiewicz [11]. Godefroid and Huth [8] proved that L_μ formulas are closed under semantic minimization, i.e., every L_μ formula can be translated to an equivalent L_μ formula (in classical logic), for which compositional checking yields the most precise answer. The translation, however, is complicated and includes several steps: transforming L_μ formulas to non-deterministic tree automata, making non-deterministic tree automata 3-valued, and translating back these automata to L_μ. Our semantic minimization procedure is more straightforward and only uses the simple tableau-based construction described in [11]. Finally, we show that our semantic minimization procedure can be extended to abstract models described as PKSs and MixTSs, thus providing a general SemanticMinimization() subroutine for the algorithm in Figure 1(a).

The rest of this paper is organized as follows: Section II outlines some preliminaries. Section III defines an automata intersection game inspired by the abstraction framework in [12]. This game is used in Section IV to prove the main result of the paper which establishes a connection between self-minimizing formulas over HTSs and disjunctive/conjunctive forms of L_μ. Section V provides a complete algorithm for thorough checking of L_μ over arbitrary abstract models including PKSs, MixTSs, and HTSs, and discusses the complexity of this algorithm. In Section VI, we present some self-minimizing fragments of CTL for HTSs. We further discuss our work and compare it to related work in Section VII. Section VIII concludes the paper. Proofs for the major theorems are available in the extended version of this paper [13].

II. Preliminaries

In this section, we provide background on modelling formalisms, temporal logics, refinement relation, and compositional and thorough semantics.

3-valued logic. We denote by **3** the 3-valued Kleene logic [14] with elements true (t), false (f), and maybe (m). The truth ordering $\leq$ of this logic is defined as $\mathsf{f} \leq \mathsf{m} \leq \mathsf{t}$, and negation as $\neg\mathsf{t} = \mathsf{f}$ and $\neg\mathsf{m} = \mathsf{m}$. An additional ordering $\preceq$ relates values based on the amount of information: $\mathsf{m} \preceq \mathsf{t}$ and $\mathsf{m} \preceq \mathsf{f}$, so that m represents the least amount of information.

Models. In what follows, we introduce different modelling formalisms that are used in this paper.

A *Kripke structure (KS)* is a tuple $K = (\Sigma, s\ , R, L, AP)$, where Σ is a set of states, $s\ \in \Sigma$ is the initial state, $R \subseteq \Sigma \times \Sigma$ is a transition relation, AP is the set of *atomic propositions*, and $L : \Sigma \to 2^{AP}$ is a labelling function. We assume KSs are total, i.e., R is left-total.

A *Partial Kripke Structure (PKS)* [1] is a KS whose labelling function L is 3-valued, i.e., $L : \Sigma \to \mathbf{3}^{AP}$. Figure 1(c) illustrates a PKS, where propositions p and q are m in state s .

An *Mixed Transition System (MixTS)* [2], [3] is a tuple $(\Sigma, s\ , R^{must}, R^{may}, L, AP)$, where Σ is a set of states, $s\ \in \Sigma$ is the initial state, $R^{must} \subseteq \Sigma \times \Sigma$ and $R^{may} \subseteq \Sigma \times \Sigma$ are *must* and *may* transition relations, respectively, AP is the set of atomic propositions, and $L : \Sigma \to \mathbf{3}^{AP}$ is a 3-valued labelling function.

A *hyper-transition system (HTS)* [4], [5], [6] is a tuple $H = (\Sigma, s\ , R^{must}, R^{may}, L, AP)$, where $R^{must} \subseteq \Sigma \times \mathcal{P}(\Sigma)$ and $R^{may} \subseteq \Sigma \times \Sigma$ are *must* and *may* transition relations, respectively, $L : \Sigma \to 2^{AP}$ is a 2-valued labelling function, and Σ, $s\ $ and AP are defined as above. Intuitively, an HTS is a MixTS with a 2-valued labelling function and must hyper-transitions. We assume HTSs and MixTSs are total, i.e., R^{may} is left-total. Figure 1(d) illustrates an HTS, where *must* and *may* transitions are represented as solid and dashed arrows, respectively. Throughout this paper, we often write relations as functions: for instance, $R^{may}(s)$ is the set $\{s' \mid (s, s') \in R^{may}\}$.

An HTS H is *concrete* if for every $s, s' \in \Sigma$, we have $s' \in R^{may}(s) \Leftrightarrow \{s'\} \in R^{must}(s)$. For every KS $K = (\Sigma, s\ , R, L, AP)$, there is an equivalent concrete HTS $H_K = (\Sigma, s\ , R^{must}, R^{may}, L, AP)$, where $R^{may} = R$ and $s' \in R(s) \Leftrightarrow \{s'\} \in R^{must}(s)$ for every $s, s' \in \Sigma$.

Temporal logics. Temporal properties are specified in the *propositional μ-calculus* L_μ [15].

Definition 1: Let Var be a set of fixpoint variables, and AP be a set of atomic propositions. The *logic* $L_\mu(AP)$ is the set of formulas generated by the following grammar:

$$\varphi ::= \mathsf{true} \mid p \mid Z \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid EX\varphi \mid \mu Z \cdot \varphi(Z)$$

where $p \in AP$, $Z \in Var$, and $\varphi(Z)$ is syntactically monotone in Z.

The derived connectives are defined as follows:

$$\begin{array}{lcl} \varphi_1 \vee \varphi_2 & = & \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ AX\varphi & = & \neg EX\neg\varphi \\ \nu Z \cdot \varphi(Z) & = & \neg\mu Z \cdot \neg\varphi(\neg Z) \end{array}$$

Any L_μ formula can be transformed into an equivalent formula in which negations are applied only to atomic propositions. Such formulas are said to be in negation normal form

$$
\begin{array}{lcl}
||\mathsf{true}||_\top e & = & \Sigma \\
||\mathsf{true}||_\bot e & = & \emptyset \\
||p||_\top e & = & \{s \mid p \in L(s)\} \\
||p||_\bot e & = & \{s \mid p \notin L(s)\} \\
||Z||_\top e & = & \overline{e(Z)} \\
||Z||_\bot e & = & e(Z) \\
||\neg\varphi||_\top e & = & ||\varphi||_\bot e \\
||\neg\varphi||_\bot e & = & ||\varphi||_\top e \\
||\varphi_1 \wedge \varphi_2||_\top e & = & ||\varphi_1||_\top e \cap ||\varphi_2||_\top e \\
||\varphi_1 \wedge \varphi_2||_\bot e & = & ||\varphi_1||_\bot e \cup ||\varphi_2||_\bot e \\
||EX\varphi||_\top e & = & ex(||\varphi||_\top e) \\
||EX\varphi||_\bot e & = & ax(||\varphi||_\bot e) \\
||\mu Z \cdot \varphi||_\top e & = & \bigcap\{S \subseteq \Sigma \mid ||\varphi||_\top e[Z \rightarrow S] \subseteq S\} \\
||\mu Z \cdot \varphi||_\bot e & = & \bigcup\{S \subseteq \Sigma \mid S \subseteq ||\varphi||_\bot e[Z \rightarrow S]\}
\end{array}
$$

Fig. 2. The semantics of L_μ.

(NNF). An L_μ formula φ is universal (resp. existential) if $\mathtt{NNF}(\varphi)$ does not contain any EX (resp. AX) operators. We write $\varphi_\forall$ (resp. $\varphi_\exists$) to denote a universal (resp. existential) formula, and write φ_{prop} when φ is a propositional formula, i.e., when φ consists only of literals, conjunctions and disjunctions.

Definition 2: [5] Let H be an HTS, φ be an L_μ formula, and $e : Var \rightarrow \mathcal{P}(\Sigma)$ be an environment. We denote by $||\varphi||^H_\top e$ the set of states in H that satisfy φ, and by $||\varphi||^H_\bot e$ the set of states in H that refute φ. The sets $||\varphi||_\top e$ and $||\varphi||_\bot e$ are defined in Figure 2, where $ex(S) = \{s \mid \exists S' \in R^{must}(s) \cdot S' \subseteq S\}$ and $ax(S) = \{s \mid \forall s' \in R^{may}(s) \cdot s' \in S\}$.

For a closed L_μ formula φ, $||\varphi||^H_\lambda e = ||\varphi||^H_\lambda e$ for any e and e and $\lambda \in \{\top, \bot\}$. Thus, e can be safely dropped when φ is closed. We also omit H when it is clear from the context. Since KSs are special cases of HTSs, the above semantics applies to them as well.

In this paper, we often express temporal formulas in the *computation tree logic* CTL [16] whose syntax is defined w.r.t. a set AP of atomic propositions as follows:

$$
\varphi \quad ::= \quad p \mid \neg p \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid EX\varphi \mid AX\varphi \mid E[\varphi U \varphi] \mid A[\varphi U \varphi] \mid E[\varphi \tilde{U} \varphi] \mid A[\varphi \tilde{U} \varphi]
$$

where $p \in AP$. The operators AU and EU are universal and existential until operators, respectively; and operators $E\tilde{U}$ and $A\tilde{U}$ are their duals, respectively. Other CTL operators can be defined from these:

$$
\begin{array}{lcllcl}
AG\varphi & = & A[\mathsf{false}\tilde{U}\varphi] & EG\varphi & = & E[\mathsf{false}\tilde{U}\varphi] \\
AF\varphi & = & A[\mathsf{true}U\varphi] & EF\varphi & = & E[\mathsf{true}U\varphi]
\end{array}
$$

CTL has a fixpoint characterization which provides a straightforward procedure for translating CTL to L_μ. Thus, the semantics of CTL over HTSs follows from Definition 2.

3-valued compositional semantics. An HTS H is *consistent* [4] if for every $s \in \Sigma$ and $S \in R^{must}(s)$, $S \cap R^{may}(s) \neq \emptyset$. Therefore, for every consistent HTS H and $\varphi \in L_\mu$, $||\varphi||_\top \cap ||\varphi||_\bot = \emptyset$, i.e., a consistent H does not satisfy $\varphi \wedge \neg\varphi$.

The semantics of L_μ over a consistent HTS H can be described as a 3-valued function $||.||^H_{\mathbf{3}} : L_\mu \times \Sigma \rightarrow \mathbf{3}$. We write $||\varphi||^H_{\mathbf{3}}(s) = \mathsf{t}$ if $s \in ||\varphi||^H_\top$, $||\varphi||^H_{\mathbf{3}}(s) = \mathsf{f}$ if $s \in ||\varphi||^H_\bot$, and $||\varphi||^H_{\mathbf{3}}(s) = \mathsf{m}$ otherwise. The value of φ in H, denoted $||\varphi||^H_{\mathbf{3}}$, is defined as $||\varphi||^H_{\mathbf{3}}(s\ )$, where s is the initial state of H. To disambiguate from an alternative semantics presented later, we refer to this 3-valued semantics of L_μ over HTSs as *compositional*.

Refinement relation. Models with 3-valued semantics are compared using ordering relations known as *refinement relations* [17].

Definition 3: [5] Let H and H be HTSs. A *refinement relation* $\rho \subseteq \Sigma \times \Sigma$ is the largest relation where $\rho(s,t)$ iff

1) $L_1(s) = L_2(t)$,
2) $\forall S \subseteq \Sigma_1 \cdot R_1^{must}(s,S) \Rightarrow \exists T \subseteq \Sigma_2 \cdot R_2^{must}(t,T) \wedge \hat{\rho}(S,T)$,
3) $\forall t' \in \Sigma_2 \cdot R_2^{may}(t,t') \Rightarrow \exists s' \in \Sigma_1 \cdot R_1^{may}(s,s') \wedge \rho(s',t')$,

where $\hat{\rho}(S,T) \Leftrightarrow \forall t' \in T \cdot \exists s' \in S \cdot \rho(s',t')$.

We say H refines H and write $H \preceq H$, if there is a refinement ρ such that $\rho(s\ , s\ )$, where s and s are the initial states of H and H, respectively.

Refinement preserves L_μ formulas [5], i.e., if $H \preceq H$, then for every $\varphi \in L_\mu$, $||\varphi||^{H_1}_{\mathbf{3}} \preceq ||\varphi||^{H_2}_{\mathbf{3}}$. Refinement can relate HTSs to KSs as well. Recall that every KS K is equivalent to a concrete HTS H_K. We say that a KS K refines an HTS H, denoted $H \preceq K$, iff $H \preceq H_K$. For an HTS H, let $\mathcal{C}[H]$ denote the set of *completions* of H, that is, the set of all KSs that refine H.

Thorough semantics and semantic minimization. Thorough semantics of L_μ over HTSs is defined w.r.t. the completions of HTSs: A formula φ is true in H under thorough semantics, written $||\varphi||^H_t = \mathsf{t}$, if it is true in all completions of H; it is false in H, written $||\varphi||^H_t = \mathsf{f}$, if it is false in all completions of H, and it is maybe in H, written $||\varphi||^H_t = \mathsf{m}$, otherwise [1].

Thorough semantics is more precise than compositional semantics [1]. That is, $||\varphi||^H_{\mathbf{3}} \preceq ||\varphi||^H_t$ for every HTS H and L_μ formula φ. A formula φ is a *positive semantic minimization* of a formula φ' if for every HTS H, $||\varphi'||^H_t = \mathsf{t} \Leftrightarrow ||\varphi||^H_{\mathbf{3}} = \mathsf{t}$, and is a *negative semantic minimization* of φ' if for every HTS H, $||\varphi'||^H_t = \mathsf{f} \Leftrightarrow ||\varphi||^H_{\mathbf{3}} = \mathsf{f}$. Further, a formula φ is called *positively self-minimizing* when it is its own positive semantic minimization, and is *negatively self-minimizing* when it is its own negative semantic minimization. A formula that is both positively and negatively self-minimizing is called *semantically self-minimizing* or *self-minimizing* for short. For instance, $AGp \wedge AGq$ is not negatively self-minimizing, because for the HTS H in Figure 1(d), $||AGp \wedge AGq||^H_{\mathbf{3}} = \mathsf{m}$ and $||AGp \wedge AGq||^H_t = \mathsf{f}$. As we show later in the paper, $AG(p \wedge q)$ is a negative semantic minimization of $AGp \wedge AGq$. Dually, $EF(p \vee q)$ is a positive semantic minimization of $EFp \vee EFq$. Since thorough semantics is defined in terms of completions of HTSs, it is desirable to define self-minimizing formulas in the same terms, via an equivalent definition, as done below.

Definition 4: An L_μ formula φ is *negatively self-minimizing* if for every HTS H, $||\varphi||^H_{\mathbf{3}} \neq \mathsf{f} \Rightarrow \exists K \in \mathcal{C}[H] \cdot K \models \varphi$, and is *positively self-minimizing* if for every HTS H, $||\varphi||^H_{\mathbf{3}} \neq \mathsf{t} \Rightarrow \exists K \in \mathcal{C}[H] \cdot K \models \neg\varphi$.

Our definitions for positive and negative semantic minimization are, respectively, the same as those for *pessimistic* and *optimistic semantic minimization* in [8].

III. An Automata Intersection Game

In this section, we define an automata intersection game inspired by the automata-based abstraction framework proposed in [12]. In this framework, both temporal formulas and abstract

models are represented as finite automata. For a formula φ, the language of its corresponding automaton $\mathcal{A}_\varphi$ is the set of KSs satisfying φ, i.e., $K \in \mathcal{L}(\mathcal{A}_\varphi)$ iff $K \models \varphi$. For an abstract model H, the language of its corresponding automaton $\mathcal{A}_H$ is the set of completions of H, i.e., $\mathcal{C}[H] = \mathcal{L}(\mathcal{A}_H)$. Viewing formulas and abstract models as automata allows us to uniformly define both (thorough) model checking and refinement checking as automata language inclusion. That is, $H \models \varphi$ iff $\mathcal{L}(\mathcal{A}_H) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$ (model checking), and $H \preceq H$ iff $\mathcal{L}(\mathcal{A}_{H_1}) \subseteq \mathcal{L}(\mathcal{A}_{H_2})$ (refinement checking) [12].

The class of automata used in [12] is known as μ-automata [11]. These automata, although very similar to non-deterministic tree automata (e.g., [18]), are more appropriate for working with branching time logics, because they precisely capture L_μ over transition systems with *unbounded* branching. We use a simplified definition of μ-automata adapted from [11], [12].

Definition 5: [11], [12] A *μ-automaton* is a tuple $\mathcal{A} = (Q, B, q\ , CH, BR, L, \Omega, AP)$, where Q is a non-empty, countable set of states called *choice* states; B is a countable set of states, disjoint from Q, called *branch* states; $q \in Q$ is the initial state; $CH \subseteq Q \times B$ is a choice relation, from choice states to branch states; $BR \subseteq B \times Q$ is a transition relation, from branch states to choice states; $L : B \to 2^{AP}$ is a labelling function mapping each branch state to a subset of atomic propositions in AP; and $\Omega : Q \to \mathbb{N}$ is an indexing function, defining a parity acceptance condition.

Unless stated otherwise, "automata" and "μ-automata" are used interchangeably in the rest of the paper. Given an infinite tree T rooted at r , a *tree run* of an automaton $\mathcal{A}$ on T is an infinite tree T' whose root is labelled with $(r\ , q\ )$. Every node of T' is labelled with either a pair (r, q) or (r, b), where r is a node from T, and q and b are respectively choice and branch states of $\mathcal{A}$. Every node (r, q) has at least one child node (r, b), where $b \in CH(q)$ and the labelling of b matches that of r. For every node (r, b) and every child r' of r in T, there exists a child (r', q') of (r, b) s.t. $q' \in BR(b)$. For every node (r, b) and every $q'' \in BR(b)$, there exists a child (r'', q'') of (r, b) s.t. r'' is a child of r in T. A *tree run T' is accepting* if on every infinite path π of T', the least value of $\Omega(q)$, for the choice states q that appear infinitely often on π, is even. An *input tree T is accepted* by $\mathcal{A}$ if there is *some* tree run of $\mathcal{A}$ on T that is accepting. The *language* of an automaton is the set of trees it accepts. For example, the language of the automaton shown in Figure 3(b) is the set of all infinite trees whose nodes are labelled by $\{p, q\}$ or $\{p, \neg q\}$. Input trees for μ-automata have arbitrary branching degrees and are not necessarily binary. For a more detailed treatment of μ-automata, reader can refer to [12]. We give a translation from HTSs to automata as follows.

Definition 6: Let $H = (\Sigma, s\ , R^{must}, R^{may}, L, AP)$ be an HTS. The automaton associated with H, $\mathcal{A}_H = (Q, B, q\ , CH, BR, L', \Omega, AP)$, has choice states $Q = \{q_i \mid s_i \in \Sigma\}$, branch states $B = \{b_{i,S} \mid s_i \in \Sigma, S \subseteq R^{may}(s_i)\}$, and the initial state q that corresponds to s . The labelling of a branch state $b_{i,S}$ is the labelling of s_i in H, i.e., $L'(b_{i,S}) = L(s_i)$. The indexing function assigns 0 to every choice state, making all choice states accepting. The transition relations are: $CH = \{(q_i, b_{i,S}) \mid \forall S' \in R^{must}(s_i) \cdot S' \cap S \neq \emptyset\}$ and $BR = \{(b_{i,S}, q_j) \mid s_j \in S\}$.

For example, the translation $\mathcal{A}_H$ of the HTS H in Figure 1(d) is shown in Figure 3(a). For every abstract model H, $\mathcal{L}(\mathcal{A}_H)$ should be equal to $\mathcal{C}[H]$. For a consistent HTS H, all of whose completions are expressible as KSs, our translation in Definition 6 guarantees that $\mathcal{L}(\mathcal{A}_H) = \mathcal{C}[H]$.

Theorem 1: Let H be a consistent HTS with the additional requirement that for every $s \in \Sigma$ and every $S \in R^{must}(s)$, we have $S \subseteq R^{may}(s)$. Then, $\mathcal{L}(\mathcal{A}_H) = \mathcal{C}[H]$.

The proof of Theorem 1 is similar to that of Lemma 1 in [12].

In [12], a game-based simulation over automata has been defined as a sufficient condition for language inclusion, i.e., if $\mathcal{A}$ is simulated by $\mathcal{A}$, then $\mathcal{L}(\mathcal{A}\ ) \subseteq \mathcal{L}(\mathcal{A}\ )$. We adapt the definition of automata simulation from [12] to define an automata intersection game. We prove that the existence of a winning strategy for this game is a sufficient condition for non-emptiness of $\mathcal{L}(\mathcal{A}\ ) \cap \mathcal{L}(\mathcal{A}\ )$. Each play in an automata intersection game is a sequence of pairs of states of the same type. Here, being of the same type means that both states are either choice states or branch states. A pair of choice (resp. branch) states is called a *choice (resp. branch) configuration*. At a choice configuration $(q\ , q\ )$, only Player I can move, choosing one branch state in $CH(q\ )$ and one in $CH(q\ )$ that match on labels. Player I's goal is to find a common path that is accepted by both $\mathcal{A}$ and $\mathcal{A}$. At a branch configuration $(b\ , b\ )$, Player II moves first and chooses any side, b or b , and any successor of that side. Player I has to respond with a successor of the other side. Intuitively, Player I wins the play if Player II cannot steer it onto a path which is not accepted by either of the automata.

Definition 7: Let $\mathcal{A}$ and $\mathcal{A}$ be automata with initial states q and q , respectively. A *$(q\ , q\ )$-game* is defined as follows:

1) (Initial) The initial configuration is $(q\ , q\ )$
2) (Choice) In a choice configuration $(q\ , q\ ) \in Q \times Q$, Player I chooses b in $CH\ (q\ )$ and b in $CH\ (q\ )$. The play continues from configuration $(b\ , b\ )$.
3) (Branch) In a branch configuration $(b\ , b\ ) \in B \times B$, the play can proceed in one of the following ways:
   a) The play ends and is a win for Player I if $L\ (b\ ) = L\ (b\ )$; it is a win for Player II otherwise.
   b) Player II chooses a 'side' $i \in \{1, 2\}$, and a choice state q_i in $BR_i(b_i)$; Player I must respond with a choice state q_j in $BR_j(b_j)$ from the other side j. The play continues from the configuration $(q\ , q\ )$.

If a finite play ends by rule 3a, the winner is as specified in that rule[1]. For an infinite play π and $i \in \{1, 2\}$, let $proj_i(\pi)$ be the infinite sequence from Q_i^ω obtained by projecting the choice configurations of π onto component i. Then, π is a win

[1]Since KSs are assumed to be total, we do not deal with finite plays in this paper. Thus, condition 3a in Definition 7 only ensures that if an infinite play π is won by Player I, then for every branch configuration (b_1, b_2) on π, $L_1(b_1) = L_2(b_2)$.

for Player I iff *proj* (π) and *proj* (π) satisfy the acceptance conditions for $\mathcal{A}$ and $\mathcal{A}$, respectively.

We say that there is an *intersection relation* $\sqcap$ between $\mathcal{A}$ and $\mathcal{A}$, written as $\mathcal{A} \sqcap \mathcal{A}$, if Player I has a winning strategy for the (q, q)-game.

Theorem 2: $\mathcal{A} \sqcap \mathcal{A}$ implies $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}) \neq \emptyset$.

By Definition 4, a formula φ is negatively self-minimizing if for every HTS H over which φ is non-false, there is a completion satisfying φ. In automata-theoretic terms, some completion of H satisfying φ exists iff $\mathcal{L}(\mathcal{A}_H) \cap \mathcal{L}(\mathcal{A}_\varphi) \neq \emptyset$. The following theorem shows that finding a winning strategy for the intersection game between an HTS automaton and a formula automaton is sufficient to show self-minimization.

Theorem 3: An L_μ formula φ is negatively self-minimizing if for every HTS H, $||\varphi||_{\mathbf{3}}^H \neq \mathsf{f} \Rightarrow \mathcal{A}_H \sqcap \mathcal{A}_\varphi$, and is positively self-minimizing if for every HTS H, $||\varphi||_{\mathbf{3}}^H \neq \mathsf{t} \Rightarrow \mathcal{A}_H \sqcap \mathcal{A}_{\neg\varphi}$.

This theorem follows from Theorem 2 and Definition 4. We use it in the next section to prove the main result of the paper.

IV. Disjunctive/Conjunctive L_μ and Self-Minimization

In this section, we introduce disjunctive L_μ and its dual, conjunctive L_μ, defined in [11], and prove that L_μ formulas in disjunctive and conjunctive forms are, respectively, negatively and positively self-minimizing.

We start by noting that arbitrary L_μ formulas may not be self-minimizing. For instance, HTS H in Figure 1(d) has completions that satisfy either AGp or AGq, but there is no completion satisfying $AGp \wedge AGq$. Thus, we cannot inductively prove that formulas of the form $\varphi \wedge \varphi$ are negatively self-minimizing (or $\varphi \vee \varphi$ positively self-minimizing). Intuitively, this is the same reason why *satisfiability* of $\varphi \wedge \varphi$ cannot be proven by structural induction. [11] proposes a syntactic form of L_μ formulas, referred to as *disjunctive* L_μ, for which satisfiability can be proven inductively. The analogy between identifying negatively self-minimizing formulas and the satisfiability problem suggests that disjunctive L_μ may be negatively self-minimizing. We prove this below.

Definition 8: [11] Let Γ be a finite set of L_μ formulas. We define $\mathtt{ref}(\Gamma) = \bigwedge_{\psi\in\Gamma} EX\psi \wedge AX \bigvee_{\psi\in\Gamma} \psi$. *Disjunctive* L_μ, denoted $L_\mu^\vee$, is the set of formulas generated by the following grammar:

$$\varphi ::= p \mid \neg p \mid Z \mid \varphi \vee \varphi \mid \varphi_1 \wedge \ldots \wedge \varphi_n \mid \sigma(Z) \cdot \varphi(Z)$$

where $p \in AP$, $Z \in Var$ and $\sigma \in \{\mu, \nu\}$; and for $\sigma(Z)\cdot\varphi(Z)$, Z occurs in $\varphi(Z)$ only positively, and does not occur in any context $Z \wedge \psi$ or $\psi \wedge Z$ for some ψ; $\varphi \wedge \ldots \wedge \varphi_n$ $(n > 1)$ is a *special conjunction*: every φ_i is either a literal (p or $\neg p$) or a formula of a form $\mathtt{ref}(\Gamma)$ for a finite set Γ of $L_\mu^\vee$ formulas, and at most one of the φ_i is of the form $\mathtt{ref}(\Gamma)$. Dually, we define *conjunctive* L_μ, denoted $L_\mu^\wedge$, consisting of all L_μ formula φ where $\mathtt{NNF}(\neg\varphi) \in L_\mu^\vee$.

Every $\varphi \in L_\mu^\vee$ can be linearly translated to a μ-automaton $\mathcal{A}_\varphi$ so that $K \in \mathcal{L}(\mathcal{A}_\varphi)$ iff $K \models \varphi$ [11]. For example, a μ-automaton $\mathcal{A}_{AGp}$ corresponding to AGp is shown in Figure 3(b) [2]. Let K be a KS over $AP = \{p, q\}$. K satisfies AGp iff all its states are labelled with $\{p, q\}$ or $\{p, \neg q\}$, i.e., unfolding K from its initial state results in an infinite tree, all of whose nodes, are labelled with $\{p, q\}$ or $\{p, \neg q\}$. Hence, the tree is accepted by $\mathcal{A}_{AGp}$, and so is K.

The formula $AG(p \wedge q) = \nu Z \cdot p \wedge q \wedge AXZ$ is in $L_\mu^\vee$, but $AGp \wedge AGq$ is not, because the conjunction is not special. A non-disjunctive formula such as $AGp \wedge AGq$ would be first written in its disjunctive form, $AG(p \wedge q)$, and then translated to a μ-automaton. Automaton $\mathcal{A}_{AG(p\wedge q)}$ is exactly the same as $\mathcal{A}_{AGp}$ but without branch state b.

Theorem 4: Every closed $L_\mu^\vee$ formula is negatively self-minimizing.

Using Definition 4, we can show by structural induction that every $L_\mu^\vee$ formula except the greatest fixpoint is negatively self-minimizing [8]. As argued in [8], a naive proof does not work for the greatest fixpoint formulas: Let $\varphi' = \nu Z \cdot \varphi(Z)$ and $||\varphi'||_{\mathbf{3}}^H \neq \mathsf{f}$. By the semantics of the greatest fixpoint, $||\varphi^i(\mathsf{t})||_{\mathbf{3}}^H \neq \mathsf{f}$ for every $i > 0$. By inductive hypothesis, for every i there is a completion K_i of H that satisfies $\varphi^i(\mathsf{t})$. While the sequence of $\varphi^i(\mathsf{t})$ converges to the fixpoint φ' on H, it is not clear whether the sequence of K_i converges to a completion of H satisfying φ'.

In our proof, we use the automata intersection game introduced in Section III. Instead of explicitly constructing a KS $K \in \mathcal{C}[H]$ satisfying φ', we prove that such a completion exists by showing a winning strategy of Player I for the game $\mathcal{A}_H \sqcap \mathcal{A}_{\varphi'}$. We sketch the proof and illustrate it by an example that uses a greatest fixpoint operator.

By inductive hypothesis, Player I has a winning strategy T^i for $\mathcal{A}_H \sqcap \mathcal{A}_{\varphi^i(\mathsf{t})}$ for every i. For a large enough i, we can convert T^i to a winning strategy T for $\mathcal{A}_H \sqcap \mathcal{A}_{\varphi'}$: Automaton $\mathcal{A}_{\varphi^i(\mathsf{t})}$ is a finite chain of unfoldings of $\mathcal{A}_{\varphi'}$, i.e., there is a morphism h which partially maps states of $\mathcal{A}_{\varphi^i(\mathsf{t})}$ to those of $\mathcal{A}_{\varphi'}$. We apply h to T^i to obtain T.

For example, let $\varphi' = AGp = \nu Z \cdot p \wedge AXZ$. For this formula, $\varphi(Z) = p \wedge AXZ$. Consider automata $\mathcal{A}_{AGp}$ and $\mathcal{A}_{\varphi^4(\mathsf{t})}$ shown in Figure 3(b) and (c) respectively. The mapping h is defined as follows: choice state $q_{\varphi^4(\mathsf{t})}$ is mapped to q_{AGp}, choice states $q_{\varphi^i(\mathsf{t})}$ to q_Z for $1 \leq i \leq 3$, and branch states $b_{l,i}$ to b_l for $l \in \{0, 1\}$ and $1 \leq i \leq 4$. State q_t and its corresponding branch states are left unmapped.

The winning strategy T^i is a function that for a choice configuration, returns a branch configuration, and for a branch configuration and a choice of Player II from either side, returns a choice state from the opposite side. We call two choice (branch) configurations (s, s) and (s, s') *indistinguishable by* h if $h(s) = h(s')$. For convenience, we extend h to pairs (s, s) of states, where s is in $\mathcal{A}_H$ and s is in $\mathcal{A}_{\varphi^i(\mathsf{t})}$ by letting $h(s, s) = (s, h(s))$. We define a strategy function T for Player I in the game $\mathcal{A}_H \sqcap \mathcal{A}_{\varphi'}$ as follows:

(i) For every choice configuration (q, q), let $T(q, q) = h(T^i(q, q'))$ for some q' in $\mathcal{A}_{\varphi^i(\mathsf{t})}$,

[2] In contrast to [12], [11], we assume that KSs are total. Therefore, every branch state in $\mathcal{A}_{AGp}$ has at least one successor.

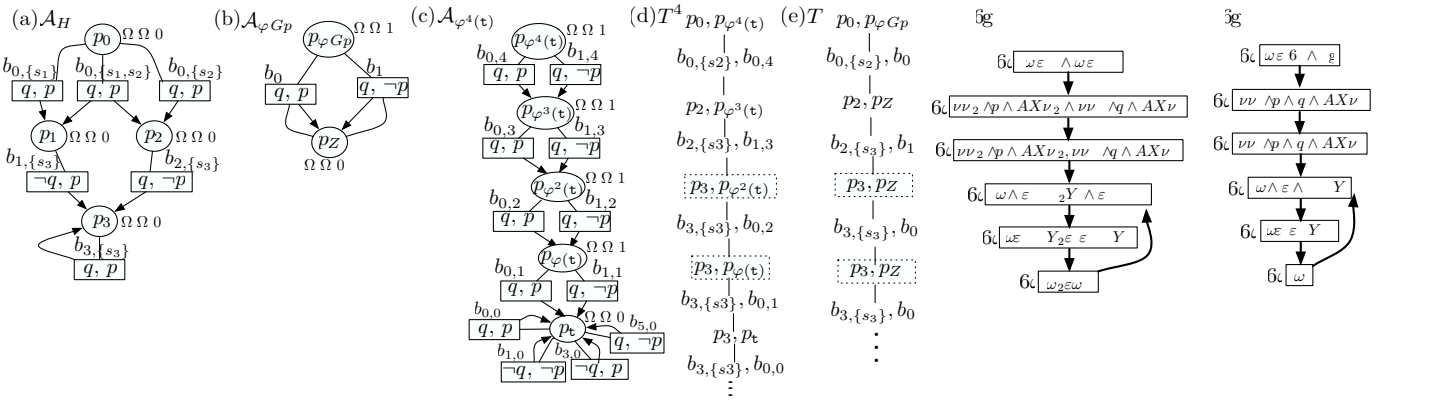


Fig. 3. Examples of automata, winning strategies, and semantic tableau: (a) automaton $\mathcal{A}_H$ corresponding to HTS H in Figure 1(d); (b) automaton $\mathcal{A}_{AGp}$; (c) automaton $\mathcal{A}_{\varphi^4(\mathfrak{t})}$; (d) winning strategy T^4 for $\mathcal{A}_H \sqcap \mathcal{A}_{\varphi^4(\mathfrak{t})}$; (e) winning strategy T for $\mathcal{A}_H \sqcap \mathcal{A}_{AGp}$; (f) the tableau associated to $AGp \wedge AGq$ [11]; and (g) extracting the disjunctive form of $AGp \wedge AGq$ from its tableau by the procedure of [11].

where $q = h(q')$;

(ii) For every branch configuration (b, b) and every choice q of Player II from $\mathcal{A}_H$, let $T(b, b, q) = h(T^i(b, b', q))$ for some b' in $\mathcal{A}_{\varphi^i(\mathfrak{t})}$, where $b = h(b')$;

(iii) For every branch configuration (b, b) and every choice q of Player II from $\mathcal{A}_{\varphi'}$, let $T(b, b, q) = T^i(b, b', q')$ for some q' and b' in $\mathcal{A}_{\varphi^i(\mathfrak{t})}$, where $q = h(q')$ and $b = h(b')$.

We first show that T is a function. Since unfoldings of $\mathcal{A}_{\varphi^i(\mathfrak{t})}$ are isomorphic, there is some T^i s.t. for every two choice (branch) configurations indistinguishable by h, T^i returns configurations indistinguishable by h as well. That is, T^i returns the same results modulo h for arbitrary $b' \in h^-(b)$ and $q' \in h^-(q)$, making T a function.

Since h is undefined for $q_{\mathfrak{t}}$, to ensure that T is a valid strategy, we need to show that in the case (ii) above, there is always a b' where $T^i(b, b', q) \neq q_{\mathfrak{t}}$. Since $\mathcal{A}_{\varphi'}$ and $\mathcal{A}_H$ are finite, such $b' \in h^-(b)$ is found for a large enough i,

Strategy T as defined above is a valid strategy function for Player I in the game $\mathcal{A}_H \sqcap \mathcal{A}_{\varphi'}$. Note that T is undefined for the inputs for which T^i modulo h is undefined. It remains to show that T is winning. Let π be any play produced by T. Play π is an infinite sequence of configurations. Automata $\mathcal{A}_H$ and $\mathcal{A}_{\varphi'}$ are finite. Thus, there must be a configuration (q, q) repeated infinitely often in π. We map π back through h^-, and distinguish two cases: (1) π is mapped back to a play π' generated by T^i s.t. infinitely many occurrences of (q, q) are mapped to a single configuration $(q, q' \neq q_{\mathfrak{t}})$ in π'. Since T^i is a winning strategy for Player I, π' is won by Player I. Since h preserves parity of state indices, π is won by Player I in T as well. (2) π is mapped back to a prefix of some π' generated by T^i s.t. infinitely many pairs of consecutive occurrences of (q, q) are mapped to two different configurations (q, q') and (q, q'') in π', where $h(q') = h(q'')$. These two configurations are different; but since they are indistinguishable by h, they have to belong to two different unfoldings in the chain $\mathcal{A}_{\varphi^i(\mathfrak{t})}$. Passing between unfoldings requires going through some state of the form $q_{\varphi^j(\mathfrak{t})}$ for some $j < i$. Since $h(q_{\varphi^j(\mathfrak{t})}) = q_Z$, there is an occurrence of (q, q_Z) for some state q of $\mathcal{A}_H$ between two consecutive occurrences of (q, q) in π. Thus, π satisfies the acceptance conditions of both $\mathcal{A}_H$ and $\mathcal{A}_{\varphi'}$ and as such, is a play won by Player I. This happens for every play generated by strategy T, and hence, strategy T is winning for Player I.

A winning strategy for $\mathcal{A}_H \sqcap \mathcal{A}_{\varphi^4(\mathfrak{t})}$, denoted T^4, and its translation T by h are shown in Figures 3(d) and (e), respectively. For this example, unfolding automaton $\mathcal{A}_{\varphi(\mathfrak{t})}$ four times is enough, because the only play produced by T^4 (shown in Figure 3(d)) visits two configurations $(q_3, q_{\varphi^2(\mathfrak{t})})$ and $(q_3, q_{\varphi(\mathfrak{t})})$ indistinguishable by h.

The following holds by duality to Theorem 4.

Theorem 5: Every closed $L_\mu^\wedge$ formula is positively self-minimizing.

Theorems 4 and 5 provide sufficient syntactic checks for identifying self-minimizing L_μ formulas that can be used in step (2) of the algorithm in Figure 1(a). Note that Theorems 4 and 5 only hold for HTSs, but not for PKSs or MixTSs. For example, $p \wedge \neg p$ is in $L_\mu^\vee$, but is not negatively self-minimizing over such models. Consider a model M with a single state in which proposition p is maybe. In M, $p \wedge \neg p$ is maybe, but this formula is false in any completion of M. In Section V, we show that by syntactically modifying disjunctive and conjunctive L_μ formulas, these formulas become negatively and positively self-minimizing over PKSs and MixTSs.

V. Thorough Checking Algorithm

In this section, we complete the thorough checking algorithm shown in Figure 1(a) by describing its subroutines IsSelfMinimizing() and SemanticMinimization(). Since we want this algorithm to work for arbitrary abstract models described as PKSs, MixTSs, or HTSs, we first need to show how disjunctive (resp. conjunctive) formulas can be made negatively (resp. positively) self-minimizing over these models.

Theorem 6: Let φ be a closed $L_\mu^\vee$ formula s.t. for every special conjunction $\psi = \psi \wedge \ldots \wedge \psi_n$ in φ, there are no literals ψ_i and ψ_j $(1 \leq i, j \leq n)$ where $\psi_i = \neg\psi_j$. Then, φ is

```
THOROUGHCHECK(M, φ)
1:  if (v: = MODELCHECK(M, φ)) ≠ maybe
2:      return v
3:  if ISSELFMINIMIZING(M, φ)
4:      return maybe
5:  v := MODELCHECK(M, SEMANTICMINIMIZATION(φ))
6:  if (v = false) return false
7:  v := MODELCHECK(M, ¬(SEMANTICMINIMIZATION(NNF(¬φ))))
8:  if (v = true) return true
9:  return maybe

ISSELFMINIMIZING(M, φ)
10: if M is a PKS or an MixTS and φ is monotone
11:     return true
12: if M is an HTS and φ ∈ L_μ^∨ ∩ L_μ^∧
13:     return true
14: return false

SEMANTICMINIMIZATION(φ)
15: convert φ to its disjunctive form φ^∨
16: replace all special conjunctions in φ^∨ containing p and ¬p with false
17: return φ^∨
```

Fig. 4. The thorough checking algorithm.

negatively self-minimizing over abstract models described as HTSs, PKSs, or MixTSs.

The above theorem can be proven using the same argument as Theorem 4. The proof of Theorem 4 fails for MixTSs and PKSs, when some special conjunction in φ is of the form $p \wedge \neg p \wedge \ldots \wedge \varphi_n$, but Theorem 6 explicitly excludes this case, and hence, remains valid. Similarly, conjunctive formulas can be made positively self-minimizing for PKSs and MixTSs with a condition dual to that in Theorem 6.

The complete thorough checking algorithm is shown in Figure 4: THOROUGHCHECK() takes an abstract model M, described as an HTS, PKS or MixTS, and an L_μ formula φ, and returns the result of thorough checking φ over M. In THOROUGHCHECK(), semantic minimization is carried out in two steps: On line 5, φ is converted to its negative, and on line 7, to its positive semantic minimization formula. If model checking the negative semantic minimization returns false, φ is false by thorough checking, too; and if model checking the positive semantic minimization returns true, φ is true by thorough checking, as well.

If the model is a PKS or an MixTS, self-minimization follows from the monotonicity of φ, and so does the check in line 10 [8], [10]. Otherwise, we check whether $\varphi \in L_\mu^\vee \cap L_\mu^\wedge$ which, by our Theorems 4 and 5, guarantees self-minimization.

In SEMANTICMINIMIZATION(), φ is first converted to its disjunctive form $\varphi^\vee$ by the tableau-based conversion in [11]. Then, any special conjunction in φ containing two literals p and $\neg p$ is replaced with false. This ensures that $\varphi^\vee$ satisfies the condition in Theorem 6. Therefore, when passed φ (resp. $\mathtt{NNF}(\neg\varphi)$) as a parameter, SEMANTICMINIMIZATION() computes a negative (resp. positive) semantic minimization of φ.

To illustrate the algorithm, recall the formula $\varphi = AGq \wedge A[pU\neg q]$ from Section I. By compositional semantics, φ is maybe over both PKS M in Figure 1(c) and HTS H in Figure 1(d). Since φ is non-monotone and non-disjunctive, it is not self-minimizing for either M or H. SEMANTICMINIMIZATION() computes φ's negative semantic minimization by first converting it into a disjunctive form $\varphi^\vee = \mu Z \cdot (q \wedge AXAGq \wedge \neg q) \vee (p \wedge q \wedge AXZ)$, and then replacing the first conjunct with false. The result is the formula $\mu Z \cdot \mathsf{false} \vee (p \wedge q \wedge AXZ)$ which is false over both M and H, meaning that φ is false by thorough checking over both models. On the other hand, the formula $AGp \wedge AGq$ is monotone and thus self-minimizing for M. However, this formula is not disjunctive and thus not self-minimizing for H. SEMANTICMINIMIZATION() computes a negative semantic minimization of this formula by converting it to its disjunctive form $AG(p \wedge q)$ which turns out to be false over H. This shows that $AGp \wedge AGq$ is false by thorough checking over H.

Complexity. Let $\varphi \in L_\mu$ and M be an abstract model. The complexity of ISSELFMINIMIZING(M, φ) is linear in the size of φ, and that of MODELCHECK(M, φ) is $O((|\varphi| \cdot |M|)^{\lfloor d/\ \rfloor +}\ )$, where d is the alternation depth of φ [19]. Thus, for the class of self-minimizing formulas, the running time of THOROUGHCHECK(M, φ) is the same as that of compositional model checking, i.e., $O((|\varphi| \cdot |M|)^{\lfloor d/\ \rfloor +}\ )$.

The complexity of SEMANTICMINIMIZATION(φ), i.e., the complexity of converting an L_μ formula φ to its disjunctive $\varphi^\vee$ or conjunctive $\varphi^\wedge$ form, is $O(2^{O(|\varphi|)})$, producing formulas of size $O(2^{O(|\varphi|)})$ [11]. Therefore, for formulas requiring semantic minimization, the running time of THOROUGHCHECK(M, φ) is $O((2^{O(|\varphi|)} \cdot |M|)^{\lfloor d/\ \rfloor +}\ )$, where d is the maximum of the alternation depths of $\varphi^\vee$ and $\varphi^\wedge$. When $\varphi^\vee$ and $\varphi^\wedge$ are alternation-free, i.e., $d = 0$, the complexity of THOROUGHCHECK(M, φ) becomes linear in the size of the abstract model, making the procedure efficient. However, we leave to future work the study of the relationships between the alternation depths of $\varphi^\vee$ and $\varphi^\wedge$ and that of φ.

VI. SELF-MINIMIZATION FOR CTL

In Section IV, we gave sufficient syntactic conditions for identifying self-minimizing L_μ formulas. Since CTL is used more often than L_μ in practice, it is useful to identify self-minimizing fragments of CTL as well. We do so by constructing grammars that generate positively/negatively self-minimizing CTL formulas.

[8] gives two grammars for negatively/positively self-minimizing formulas. Using our results on self-minimization checks of disjunctive/conjunctive L_μ, we extend these grammars as shown in Figure 5: φ^{neg} generates negatively and φ^{pos} generates positively self-minimizing formulas. The new constructs $A[\varphi_{prop} U \varphi^{neg}]$ and $A[\varphi^{neg} \tilde{U} \varphi_{prop}]$ added to the φ^{neg} grammar include formulas such as AGp and $A[pUq]$ that are negatively self-minimizing by Theorem 4. The construct $E[\varphi^{pos} U \varphi_{prop}]$ added to the φ^{pos} grammar includes, for instance, EFp that is positively self-minimizing by Theorem 5. Clearly, these grammars still do not capture the entire CTL which is not surprising because CTL is not closed under semantic minimization [8].

The notion of self-minimization in our grammars works only for HTSs. For example, φ^{neg} can generate $p \wedge \neg p$ which is not positively self-minimizing for either PKSs or MixTSs. To extend our grammars to these formalisms, we could restrict the grammar rules as in [9], [8] for propositional formulas, so that they do not produce non-monotone formulas.

$$\varphi^{neg} ::= p \mid \neg p \mid \varphi^{neg} \vee \varphi^{neg} \mid \varphi^{neg}_{\exists} \wedge \varphi^{neg}_{\exists} \mid \texttt{ref}(\Gamma^{neg}) \mid E[\varphi^{neg}_{\exists} U \varphi^{neg}] \mid E[\varphi^{neg}_{\exists} \tilde{U} \varphi^{neg}_{\exists}] \mid A[\varphi_{prop} U \varphi^{neg}] \mid A[\varphi^{neg} \tilde{U} \varphi_{prop}]$$

$$\varphi^{pos} ::= p \mid \neg p \mid \varphi^{pos} \wedge \varphi^{pos} \mid \varphi^{pos}_{\forall} \vee \varphi^{pos}_{\forall} \mid \widehat{\texttt{ref}}(\Gamma^{pos}) \mid E[\varphi^{pos} U \varphi_{prop}] \mid E[\varphi_{prop} \tilde{U} \varphi^{pos}] \mid A[\varphi^{pos}_{\forall} U \varphi^{pos}_{\forall}] \mid A[\varphi^{pos}_{\forall} \tilde{U} \varphi^{pos}]$$

Fig. 5. The grammar φ^{neg} (resp. φ^{pos}) generates negatively (resp. positively) self-minimizing subsets of CTL: Γ^{neg} (resp. Γ^{pos}) is a finite set of formulas generated by φ^{neg} (resp. φ^{pos}), and $\widehat{\texttt{ref}}$ is the dual of the operator $\texttt{ref}$.

VII. Related Work and Discussion

The problem of thorough checking for propositional logic was considered by [9] which proposed an efficient BDD-based algorithm for semantic minimization of propositional formulas. [1], [7] studied complexities and lower bounds for thorough checking of various temporal logics. [10] proposed self-minimizing checks for CTL, and [8] extended those checks to L_μ, and further, studied semantic minimization of various temporal logics.

In [8], a series of conversions between tree-automata and L_μ formulas is used to show that a semantic minimization of an L_μ formula can be computed in exponential time. This approach is hard to implement because it uses non-deterministic tree automata whose states have *unbounded* arities. The method proposed in [11] for translating an L_μ formula to its disjunctive form has the same complexity but is easier to implement, because it uses μ-automata instead– in this kind of automata, the number of successors of each state can be obtained from the structure of the formula.

As an example, the process of transforming $AGp \wedge AGq$ into its disjunctive form $AG(p \wedge q)$ was illustrated in Figures 3(e) and (f). The tableau for $AGp \wedge AGq$, constructed based on the L_μ proof rules of [11], is shown in Figure 3(e). The disjunctive form of $AGp \wedge AGq$ is constructed by traversing this tableau from its leaves to the top and labelling each node with a formula according to the procedure of [20] (see Figure 3(f)). Similar tableau methods were used in [21] for automata-based model checking of formulas whose conjunctions are restricted to having at most one conjunct with fixpoint variables.

In our paper, we only considered HTSs with 2-valued labels. This is in contrast to the HTSs in [4], [5] where states have 3-valued labels. Following [12], HTSs with 3-valued labels can be translated to ours. If the resulting HTSs satisfy the conditions in Theorem 1, then our results apply to the more general HTSs of [4], [5] as well.

VIII. Conclusion and Future Work

In this paper, we proved that disjunctive L_μ and conjunctive L_μ are, respectively, negatively and positively self-minimizing over HTSs. We base our proof on an automata intersection game. Our results provide a simple syntactic check for identifying self-minimizing formulas. For such formulas, thorough checking is as cheap as compositional model checking. We also proposed an algorithm for semantic minimization of L_μ and showed that its complexity is linear in the size of abstract models for L_μ formulas with alternation-free disjunctive and conjunctive forms.

In [7], it was shown that the complexity of thorough checking for the class of *persistence properties* [22], i.e., properties recognizable by co-Buchi automata, is also linear in the size of the abstract model. Studying the relationships between persistence properties and L_μ formulas with alternation-free disjunctive and conjunctive forms is left for future work.

Dams and Namjoshi [12] envisioned that viewing abstract models as μ-automata can open up many important connections between abstraction and automata theory. We believe that our work establishes one such connection, paving the way for further research on automata-based approaches to abstraction.

Acknowledgment. We thank Mehrdad Sabetzadeh for his help in improving the presentation of this paper. We thank the anonymous referees for their useful comments. Financial support was provided by NSERC and MITACS.

References

[1] G. Bruns and P. Godefroid, "Generalized model checking: Reasoning about partial state spaces," in *CONCUR*, ser. LNCS, vol. 1877, 2000, pp. 168–182.
[2] D. Dams, R. Gerth, and O. Grumberg, "Abstract interpretation of reactive systems," *ACM TOPLAS*, vol. 2, no. 19, pp. 253–291, 1997.
[3] R. Cleaveland, S. P. Iyer, and D. Yankelevich, "Optimality in abstractions of model checking," in *SAS*, ser. LNCS, vol. 983, 1995, pp. 51–63.
[4] L. de Alfaro, P. Godefroid, and R. Jagadeesan, "Three-valued abstractions of games: Uncertainty, but with precision," in *LICS*, 2004, pp. 170–179.
[5] S. Shoham and O. Grumberg, "Monotonic abstraction-refinement for ctl," in *TACAS*, ser. LNCS, vol. 2988, 2004, pp. 546–560.
[6] K. Larsen and L. Xinxin, "Equation solving using modal transition systems," in *LICS*, 1990.
[7] P. Godefroid and R. Jagadeesan, "Automatic abstraction using generalized model-checking," in *CAV*, ser. LNCS, vol. 2404, 2002, pp. 137–150.
[8] P. Godefroid and M. Huth, "Model checking vs. generalized model checking: Semantic minimizations for temporal logics," in *LICS*, 2005, pp. 158–167.
[9] T. Reps, A. Loginov, and S. Sagiv, "Semantic minimization of 3-valued propositional formulae," in *LICS*, 2002.
[10] A. Gurfinkel and M. Chechik, "How thorough is thorough enough," in *CHARME*, ser. LNCS, vol. 3725, 2005, pp. 65–80.
[11] D. Janin and I. Walukiewicz, "Automata for the modal mu-calculus and related results," in *MFCS*, 1995, pp. 552–562.
[12] D. Dams and K. Namjoshi, "Automata as abstractions," in *VMCAI*, 2005, pp. 216–232.
[13] S. Nejati, M. Gheorghiu, and M. Chechik, "Thorough checking revisited." U of Toronto, Tech. Rep, CSRG-540, 2006.
[14] S. Kleene, *Introduction to Metamathematics*. New York: Van Nostrand, 1952.
[15] D. Kozen, "Results on the propositional μ-calculus," *TCS*, vol. 27, pp. 334–354, 1983.
[16] E. Clarke, E. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM TOPLAS*, vol. 8, no. 2, pp. 244–263, 1986.
[17] K. Larsen and B. Thomsen, "A modal process logic," in *LICS*, 1988, pp. 203–210.
[18] E. Emerson and C. S. Jutla, "Tree automata, mu-calculus and determinacy," in *FOCS*, 1991, pp. 368–377.
[19] A. Browne, E. M. Clarke, S. Jha, D. E. Long, and W. Marrero, "An improved algorithm for the evaluation of fixpoint expressions," *TCS*, vol. 178, no. 1–2, pp. 237–255, 1997.
[20] I. Walukiewicz, "Notes on the propositional μ-calculus: Completeness and related results," BRICS, NS-95-1, Tech. Rep., 1995.
[21] G. Bhat, R. Cleaveland, and A. Groce, "Efficient model checking via buchi tableau automata," in *CAV*, 2001, pp. 38–52.
[22] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

Optimizations for LTL Synthesis

Barbara Jobstmann and Roderick Bloem
Graz University of Technology

Abstract— **We present an approach to automatic synthesis of specifications given in Linear Time Logic. The approach is based on a translation through universal co-Büchi tree automata and alternating weak tree automata [1]. By careful optimization of all intermediate automata, we achieve a major improvement in performance.**

We present several optimization techniques for alternating tree automata, including a game-based approximation to language emptiness and a simulation-based optimization. Furthermore, we use an incremental algorithm to compute the emptiness of nondeterministic Büchi tree automata. All our optimizations are computed in time polynomial in the size of the automaton on which they are computed.

We have applied our implementation to several examples and show a significant improvement over the straightforward implementation. Although our examples are still small, this work constitutes the first implementation of a synthesis algorithm for full LTL. We believe that the optimizations discussed here form an important step towards making LTL synthesis practical.

I. Introduction

Writing both a specification and an implementation and subsequently checking whether the latter satisfies the former seems wasteful. A much more attractive approach would be to automatically construct the implementation from the specification, leaving the designer with only the task of ensuring that the specification describes the intended behavior. The benefit is even more pronounced when one takes into effect the cost for debugging the manual implementation, and of redesigning it when the specification changes.

LTL synthesis was proposed in [2]. The key to the solution is the observation that a program with input signals I and output signals O can be seen as a complete Σ-labeled D-tree with $\Sigma = 2^O$ and $D = 2^I$: the label of node $t \in D^*$ gives the output after input sequence t. The solution proposed in [2] is to build a nondeterministic Büchi word automaton for the specification and then to convert this automaton to a deterministic Rabin automaton that recognizes all complete trees satisfying the specification. A witness to the nonemptiness of the automaton is an implementation of the specification. A specification is called *realizable* if such an implementation exists.

There are two reasons that this approach has not been followed by an implementation. The first reason is that synthesis of LTL properties is 2EXPTIME-complete [3]. The second is that the solution uses an intricate determinization construction [4] that is hard to implement and very hard to optimize. The first reason should not prevent one from implementing the approach. After all, the bound is a lower bound and a manual implementation is also subject to it. (Cf. [5].) Thus, the worst case complexity of verifying the specification on a manual implementation is also 2EXPTIME in terms of the (full) specification. In combination with the second reason, however, the argument gains strength. For many specifications, a doubly-exponential blow up is not necessary, but can only be avoided through careful use of optimization techniques, which is hard to achieve in combination with Safra's algorithm.

Kupferman and Vardi [1] recently proposed an alternative to this approach. Starting from a specification φ over they generate, through the nondeterministic Büchi word automaton for $\neg\varphi$, a universal co-Büchi tree automaton that accepts all trees satisfying φ. From this automaton they construct an alternating weak tree automaton accepting at least one (regular) tree satisfying φ (or none, if φ is not realizable). Finally, the alternating automaton is converted to a nondeterministic Büchi tree automaton with the same language. A witness for the nonemptiness of this automaton is an implementation of φ. The approach is applicable to any linear logic that is closed under negation and can be compiled to nondeterministic Büchi word automaton.

This approach allows for optimizations at all steps. First of all, to generate the nondeterministic Büchi word automaton, we use the optimizations present in Wring [6]. The conversion to the universal co-Büchi tree automaton is relatively simple. If we consider the universal co-Büchi tree automaton as a game between the environment (which drives I) and the system (which drives O), states that are winning for the environment represent unrealizable specifications and can be removed. On the weak alternating tree automaton that is created next, we can perform the same optimization. Furthermore, we extend the concept of simulation to alternating tree automata and use it to optimize the automaton. Next,we compute the states of the nondeterministic automaton in a breadth-first manner and compute the game at every step. Thus, we may avoid expanding many of the states of the nondeterministic automaton. Finally, we use a simulation-based optimization to minimize the size of the resulting finite state machine. As suggested in [1], we perform the construction of the weak alternating automaton and the nondeterministic Büchi automaton incrementally. We build an increasingly large part of the weak alternating automaton and reuse results obtained using the smaller automata for the larger ones.

Our tool, Lily (Linear Logic Synthesizer), takes as input an LTL formula and a partition of the atomic propositions into input and output signals. If the specification is realizable, it delivers a Verilog file as output. We present an experimental evaluation of our implementation in Section V. Previous work on LTL synthesis focuses on restricted subsets of LTL [7], [8], [9]. Our implementation is the first to handle the complete language. It does not impose any syntactic requiren

This work was supported in part by the European Commission under contract 507219 (Prosyd).

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

the specification.

The flow of the paper is as follows. In the next section, we will introduce the necessary concepts. In Section III we describe a game-based and a simulation-based optimization that can be used on any tree automaton. In Section IV, we recall the construction of Kupferman and Vardi [1] and discuss how it can be implemented efficiently. We show experimental results and conclude in Section V.

II. Preliminaries

We assume that the reader is familiar with the μ-calculus and linear time temporal logic (LTL). (See [10].) We will use LTL to specify the behavior of a system. Properties will use the set $I \cup O$ of atomic propositions, where I and O are disjoint sets denoting the input and output signals, respectively.

A Σ-labeled D-*tree* is a tuple (T, τ) such that $T \subseteq D^*$ is prefix-closed and $\tau : T \to \Sigma$. The tree is *complete* if $T = D^*$. The set of all Σ-labeled D-trees is denoted by $T_{\Sigma,D}$.

We will use Σ-labeled D-trees to model programs with input alphabet D and output alphabet Σ. In order to establish a link with the specification, we will assume that $D = 2^I$ and $\Sigma = 2^O$. Thus, a path of a Σ-labeled D-tree can be seen as a word over $(\Sigma \cup D)^\omega$: we merge the label of the node with the direction edge following it in the path. Given a word language $L \in (\Sigma \cup D)^\omega$, let $\Lambda(L) \subseteq T_{\Sigma,D}$ be the set of trees T such that all paths of T are in L. For a word automaton A we will write $\Lambda(A)$ for $\Lambda(L(A))$. Similarly, we will write $\Lambda(\varphi)$ for the set of trees T such that every path of T satisfies the LTL formula φ.

A *Moore machine* with output alphabet Σ and input alphabet D is a tuple $M = (\Sigma, D, S, s_0, f, g)$ such that S is a finite set of states, $s_0 \in S$ is the initial state, $f : S \times D \to S$ is the transition function, and $g : S \to \Sigma$ is the output function. We extend f to the domain $S \times \Sigma^*$ in the usual way. The *input/output language* of M is $L(M) = \{\pi \in (\Sigma \cup D)^\omega \mid \pi = (\sigma_0 \cup d_0, \sigma_1 \cup d_1, \dots), \sigma_i = g(f(q_0, d_0 ... d_{i-1}))\}$.

Every Moore machine corresponds to a complete Σ-labeled D-tree for which every node $t \in D^*$ is labeled with $g(f(q_0, t))$. Thus, every tree language $L \subseteq T_{\Sigma,D}$ defines a set $\mathcal{M}(L)$ of Moore machines: those machines M for which $\Lambda(L(M)) \in L$. (Note that not every tree can be defined by a Moore machine and thus there are tree languages L for which $\bigcup\{\Lambda(L(M)) \mid M \in \mathcal{M}(L)\} \neq L$.)

An *alternating tree automaton* for Σ-labeled D-trees is a tuple $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ such that Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \to 2^{2^{D \times Q}}$ is the *transition relation* (an element $C \in 2^{D \times Q}$ is a *transition*) and $\alpha \subseteq Q$ is the *acceptance condition*. We denote by A^q, for $q \in Q$, the automaton A with the initial state q.

A run (R, ρ) of A on a Σ-labeled D-tree (T, τ) is a $T \times Q$-labeled $\mathbb{N}$-tree satisfying the following constraints:

1) $\rho(\varepsilon) = (\varepsilon, q_0)$.
2) If $r \in R$ is labeled (t, q), then there is a set $\{(d_1, q_1), \dots, (d_k, q_k)\} \in \delta(q, \tau(t))$ such that r has k children labeled $(t \cdot d_1, q_1), \dots, (t \cdot d_k, q_k)$.

Intuitively, the nodes of the run on a tree T are labeled with pairs (t, q) meaning that A is in state q in node t of T. Because A is alternating, for a given t there can be multiple q_i and nodes labeled (t, q_i) in R. The automaton starts at the root node in state q_0. If it is in state q in node t of T, and t is labeled σ, then $\delta(q, \sigma)$ tells A what to do next. The automaton can nondeterministically choose a transition $C \in \delta(q, \sigma)$. Then, for all $(d', q') \in C$, A moves to node $t \cdot d'$ in state q'. (The transition relation $\delta(q, \sigma)$ can be considered as a DNF formula over $D \times Q$.) Note that there are no runs with a node (t, q) for which $\delta(q, \tau(t)) = \emptyset$. On the other hand, a run that visits a node t needs not visit all of its children; there are no restrictions on the subtrees rooted in a node that is not visited. In particular, a node (t, q) such that $\delta(q, \tau(t)) = \{\emptyset\}$ does not have any children and there are no restrictions on the subtree rooted in t.

We have two acceptance conditions: Büchi and co-Büchi. A run (R, ρ) of a Büchi (co-Büchi) automaton is accepting if all *infinite* paths of (R, ρ) have infinitely many states in α (only finitely many states in α, resp.). The language $L(A)$ of A is the set of trees for which there exists an accepting run.

An alternating tree automaton induces a graph. The states of the automaton are the nodes of the graph and there is an edge from q to q' if (d', q') occurs in $\delta(q, \sigma)$ for some $\sigma \in \Sigma$ and $d' \in D$. A Büchi automaton is *weak* if each strongly connected component (SCC) contains either only states in α or only states not in α.

An automaton is *universal* if $|\delta(q, \sigma)| = 1$. A universal automaton has at most one run for a given input. An automaton is *nondeterministic* if for all $q \in Q, \sigma \in \Sigma, C \in \delta(q, \sigma)$ and $(d_i, q_i), (d_j, q_j) \in C$ we have $d_i = d_j$ implies $q_i = q_j$. That is, the automaton can only send one copy in each direction and every run is isomorphic to the input tree. An automaton is *deterministic* if it is both universal and nondeterministic.

An automaton is a *word automaton* if $|D| = 1$. In that case, we can leave out D altogether.

We will abbreviate alternating/nondeterministic/universal/deterministic Büchi/co-Büchi/weak tree/word automaton as a three letter acronym: A/N/U/D B/C/W T/W.

III. Simplifying tree automata

In this section we discuss two optimizations that can be used for any tree automaton.

A. Simplification Using Games

We define a sufficient (but not necessary) condition for language emptiness of A^q. Our heuristic views the alternating automaton as a game which is played in rounds. In each round, starting at a state q, the protagonist decides the label $\sigma \in \Sigma$ and a set $C \subseteq \delta(q, \sigma)$ and the antagonist chooses a pair $(d, q') \in C$. The next round starts in q'. If $\delta(q, \sigma)$ or C are empty the play is finite and the player who has to choose from an empty set loses the game. If a play is infinite the winner is determined by the acceptance condition. For an ABT (ACT), the protagonist wins the play if the play visits the set of accepting states α infinitely often (only finitely often, resp.). A *strategy* s maps a finite sequence of states $q_0, \dots, q_k$ to a set $C \subseteq \delta(q_k, \sigma)$ for some a label $\sigma \in \Sigma$. A play $q_1 \cdot q_2 \cdot \dots$ adheres to a strategy s if for every k, $s(q_0, \dots, q_k$

implies that there is a pair $(d, q_{k+1}) \in C$. The game A^q is *won* (and q is *winning*) if there is a strategy such that all plays starting at q that adhere to the strategy are won. The set of all winning states is the *winning region*.

If the game is lost, $L(A^q)$ is empty. In the case of an NBT or NCT the converse holds as well, but in the alternating case it does not: A counterexample is a word automaton such that (1) $\delta(q_0, \sigma) = q_1 \wedge q_2$ for all σ, (2) $L(A^{q_1}) \cap L(A^{q_2}) = \emptyset$, and (3) the games A^{q_1} and A^{q_2} are won. (Computing a necessary and sufficient condition in polynomial time is not possible as it would give us an EXPTIME algorithm for deciding realizability.)

The game is computed as follows. For $S \subseteq Q$, let

$$\langle \mathsf{P} \rangle \mathsf{X}(S) = \{q \mid \exists \sigma \in \Sigma, C \in \delta(q, \sigma) \forall (d, q') \in C : q' \in S\},$$
$$W_B(S) = \nu Y. \langle \mathsf{P} \rangle \mathsf{X}(\mu Z. Y \wedge (S \vee \langle \mathsf{P} \rangle \mathsf{X}\ Z)), \text{ and}$$
$$W_C(S) = \mu Y. \langle \mathsf{P} \rangle \mathsf{X}(\nu Z. Y \vee (S \wedge \langle \mathsf{P} \rangle \mathsf{X}\ Z)).$$

In an ABT (ACT) with acceptance condition α, we can discard the states outside of $W_B(\alpha)$ ($W_C(\alpha)$, resp.).

Theorem 1: Given an ABT (ACT) $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, let $W = W_B(\alpha)$. ($W = W_C(\alpha)$, resp.) If $q_0 \in W$, let the ABT (ACT) $A' = (\Sigma, D, Q', q_0, \delta', \alpha')$ with $Q' = Q \cap W$, $\alpha' = \alpha \cap W$, and $\delta'(q, \sigma) = \{C \in \delta(q, \sigma) \mid \forall (d, q') \in C, q \in W\}$. If $q_0 \notin W$, let A' consist of a single non-accepting state.

We have $L(A^q) = L(A'^q)$ for all $q \in Q'$ and thus $L(A) = L(A')$. □

B. Simplification Using Simulation Relations

The second optimization uses (direct) simulation minimization on alternating tree automata. Our construction generalizes that for alternating word automata [11], [12], [13].

Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. The *direct simulation relation* $\preceq \subseteq Q \times Q$ is the largest relation such that $u \preceq v$ implies that (1) $u \in \alpha \rightarrow v \in \alpha$ and (2) $\forall \sigma \in \Sigma, C_u \in \delta(u, \sigma)\ \exists C_v \in \delta(v, \sigma)\ \forall d' \in D, (d', v') \in C_v\ \exists (d', u') \in C_u : u' \preceq v'$.

If $u \preceq v$, we say that u is *simulated by* v. If additionally, $u \succeq v$, we say that u and v are *simulation equivalent*, denoted $u \simeq v$.

Lemma 2: If $u \preceq v$ then $L(A^u) \subseteq L(A^v)$. □

The following theorems are tree-automaton variants of those presented in [13] for optimizing alternating word automata. The first theorem allows us to restrict the state space of an ABT to a set of representatives of every equivalence class under $\simeq$.

Theorem 3: Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT, let $u, v \in Q$, and suppose $u \simeq v$. Let $A' = (\Sigma, D, Q \setminus \{u\}, q_0', \delta', \alpha)$, where $q_0' = v$ if $q_0 = u$ and $q_0' = q_0$ otherwise, and δ' is obtained from δ by replacing u by v everywhere. Then, $L(A) = L(A')$. □

The following two theorems allow us to simplify the transition relation of an ABT. The first theorem tells us that we can drop states from a transition if they are not minimal with respect to the simulation relation and the second theorem tells us that we can drop entire transitions if there are other transitions that allow for a larger language.

Theorem 4: Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. For $C \subseteq D \times Q$, let $C' = \{(d, u) \in C \mid \neg \exists v : v \neq u, v \preceq u, (d, v) \in C\}$. Let $A' = (\Sigma, D, Q, q_0, \delta', \alpha)$, where for all q and σ we have $\delta'(q, \sigma) = \{C' \mid C \in \delta(q, \sigma)\}$. We have $L(A) = L(A')$. □

Theorem 5: Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. Suppose $C, C' \in \delta(q, \sigma)$, $C \neq C'$, and for all d and $(d, q') \in C'$ there is a $(d, q) \in C$ such that $q \preceq q'$. Let $A = (\Sigma, D, Q, q_0, \delta', \alpha)$ be an ABT for which δ' equals δ except that $\delta'(q, \sigma) = \delta(q, \sigma) \setminus C$. We have $L(A) = L(A')$. □

We can simplify an ABT by repeated application of the last two theorems and removal of states that are no longer reachable from the initial state. The simulation relation can be computed in polynomial time, as can the optimizations. (Application of the theorems does not alter the simulation relation.)

IV. Optimizations for Synthesis

A. Synthesis Algorithm

The goal of synthesis is to find a Moore machine M implementing an LTL specification φ (or to prove that no such M exists). Our approach follows that of [1], introducing optimizations that make synthesis much more efficient. The flow is as follows.

1) Construct an NBW A_{NBW} with $L(A_{\text{NBW}}) = \{w \in (\Sigma \cup D)^\omega \mid w \not\models \varphi\}$. Let n' be the number of states of A_{NBW}. Note that in the worst case n' is exponential in $|\varphi|$ [14].
2) Construct a UCT A_{UCT} with $L(A_{\text{UCT}}) = T_{\Sigma,D} \setminus \Lambda(A_{\text{NBW}}) = \Lambda(\varphi)$. Let n be the number of states of A_{UCT}; we have $n \leq n'$,
3) Perform the following steps for increasing k, starting with $k = 0$.
   a) Construct an AWT $A_{\text{AWT}k}$ such that $L(A_{\text{AWT}k}) \subseteq L(A_{\text{UCT}})$ and $A_{\text{AWT}k}$ has at most $n \cdot k$ states.
   b) Construct an NBT $A_{\text{NBT}k}$ such that $L(A_{\text{NBT}k}) = L(A_{\text{AWT}k})$; $A_{\text{NBT}k}$ has at most $(k+1)^{2n}$ states.
   c) Check for the nonemptiness of $L(A_{\text{NBT}k})$. If the language is nonempty, proceed to Step 4.
   d) If $k = 2n^{2n+2}$, stop: φ is not realizable. Otherwise, proceed with the next iteration of the loop. (The bound on k follows from [15].)
4) Compute a witness for the nonemptiness of $A_{\text{NBT}k}$ and convert it to a Moore machine.

If the UCT constructed in Step 2 is weak, synthesis is much simpler: we complement the acceptance condition of A_{UCT} turning it into a UWT, a special case of an AWT. Then, we convert the UWT into an NBT A_{NBT} as in Step 3b. If $L(A_{\text{NBT}})$ is nonempty, the witness is a Moore machine satisfying φ, if it is empty, φ in unrealizable. In this case, we avoid increasing k and the size of the NBT is at most 2^{2n}.

It turns out that in practice, for realizable specifications, the algorithm terminates with very small k, often around three. It should be noted that it is virtually impossible to prove the specification unrealizable using this approach, because of the high bound on k. The one exception is if the UCT is weak, because in that case we avoid the dependence on k altogether, as explained above.

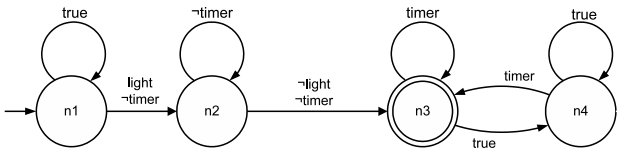


Fig. 1. NBW for $\neg\varphi = \mathsf{G}(\mathsf{F}(\text{timer})) \wedge \mathsf{F}(\text{light} \wedge (\neg\text{light}\,\mathsf{R}\,\neg\text{timer}))$

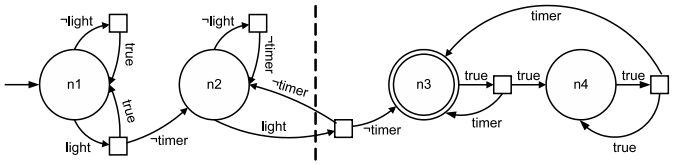


Fig. 2. UCT for $\varphi = \mathsf{G}(\mathsf{F}(\text{timer})) \rightarrow \mathsf{G}(\text{light} \rightarrow (\text{light}\,\mathsf{U}\,\text{timer}))$

In the following, we will describe the individual steps, discuss the optimizations that we use at every step, and show how to reuse information gained in one iteration of the loop for the following iterations.

B. NBW & UCT

We use Wring [6] to construct a nondeterministic generalized Büchi automaton for the negation of the specification. We then use the classic counting construction and the optimizations available in Wring to obtain a small NBW A_{NBW} with $L(A_{\text{NBW}}) = (D \cup \Sigma)^\omega \setminus L(\varphi)$.

We construct a UCT A_{UCT} over Σ-labeled D-trees with $L(A_{\text{UCT}}) = \Lambda((\Sigma \cup D)^\omega \setminus L(A_{\text{NBW}}))$.

Definition 6: [1] Given an NBW $A_{\text{NBW}} = (\Sigma, D, Q, q_0, \delta, \alpha)$, let UCT $A_{\text{UCT}} = (\Sigma, D, Q, q_0, \delta', \alpha)$, with for every $q \in Q$ and $\sigma \in \Sigma$

$$\delta'(q,\sigma) = \big\{\{(d,q') \mid d \in D, q' \in \delta(q, d \cup \sigma)\}\big\}.$$

□

We have $L(A_{\text{UCT}}) = T_{\Sigma,D} \setminus \Lambda(A_{\text{NBW}})$.

We can reduce the size of $L(A_{\text{UCT}})$ using game-based simulation and Theorem 1. Optimizing the UCT reduces the time spent optimizing the AWT and, most importantly, it may make the UCT weak, which means that we avoid the expensive construction of the AWT discussed in the next section. Because the UCT is small in comparison to the AWT and the NBT, optimization comes at a small cost.

Specifications are often of the form $\varphi \rightarrow \psi$, where φ is an assumption on the environment and ψ describes the allowed behavior of the system. When the system assertion ψ has been violated, it may still be possible to satisfy the specification by a violation of the environment assumption. However, since the system cannot control the environment, states that check that φ is violated once the system assertion ψ has been violated are not necessary. Such states, among others, are removed by the game-based optimization.

Example 7: Let $\varphi = \mathsf{G}\,\mathsf{F}\,\text{timer} \rightarrow \mathsf{G}(\text{light} \rightarrow (\text{light}\,\mathsf{U}\,\text{timer}))$. This formula is part of the specification of a traffic light controller and states that if the timer signal is set regularly, the light does not make a transition to zero unless the timer is high. The atomic propositions are partitioned into $I = \{\text{timer}\}$ and $O = \{\text{light}\}$. Fig. 1 shows a minimal NBW A_{NBW} accepting all words in $\neg\varphi$. The edges in the figure (and in the implementation) are labeled with cubes over the set of atomic propositions $I \cup O$. (A cube is a conjunct consisting of possibly negated atomic propositions.) An edge labeled with the cube c summarizes a set of edges, each labeled with a letter $w \subseteq I \cup O$ that is compatible with c.

The UCT A_{UCT} that accepts all 2^O-labeled 2^I-trees not in $T(A_{\text{NBW}})$ is shown in Fig. 2. Circles denote states and boxes denote transitions. We label edges starting at circles with cubes over O ($\Sigma = 2^O$)) and edges from boxes with cubes over I ($D = 2^I$). The transition corresponding to a box C consists of all pairs (d, q) for which there is an edge from C to q such that d satisfies the label on the edge. In particular, if d satisfies none of the labels, the branch in direction d is finite, e.g., in state n_2 with light=0 and timer=1. Recall that finite branches are accepting.

Even though the NBW is optimized, the UCT is not minimal: The tree languages $L({A_{\text{UCT}}}^{n3})$ and $L({A_{\text{UCT}}}^{n4})$ are empty. Our algorithm finds both states and replaces them by transitions to false, removing the part of A_{UCT} to the right of the dashed line. Note that the optimizations cause the automaton to become weak. □

C. AWT

From the automaton A_{UCT} we construct an AWT $A_{\text{AWT}k}$ such that $L(A_{\text{AWT}k}) \subseteq L(A_{\text{UCT}})$

Definition 8: [1] Let $A_{\text{UCT}} = (\Sigma, D, Q, q_0, \delta, \alpha)$, let $n = |Q|$ and let $k \in \mathbb{N}$. Let $[k]$ denote $\{0, \ldots, k\}$. We construct $A_{\text{AWT}k} = (\Sigma, D, Q', q_0', \delta', \alpha')$ with

$$\begin{aligned}
Q' &= \{(q,i) \in Q \times [k] \mid q \notin \alpha \text{ or } i \text{ is even}\},\\
q_0' &= (q_0, k),\\
\delta'((q,i),\sigma) &= \big\{\{(d_1,(q_1,i_1)),\ldots,(d_k,(q_k,i_k))\} \mid \\
&\qquad \{(d_1,q_1),\ldots,(d_k,q_k)\} \in \delta(q,\sigma),\\
&\qquad i_1,\ldots,i_k \in [i], \forall j : (q_j,i_j) \in Q'\big\}\\
\alpha' &= Q \times \{1,3,\ldots,2k-1\}.
\end{aligned}$$

We call i the *rank* of an AWT state (q, i). □

If $k = 2n^{n+2}$ we have $L(A_{\text{AWT}k}) = \emptyset$ implies $L(A_{\text{UCT}}) = \emptyset$ [1], [15].

We improve this construction in three ways: by using games, by merging directions, and by using simulation relations.

1) Game Simulation: We can use Theorem 1 to remove states from $A_{\text{AWT}k}$.

Example 9: Consider the UCT in Fig. 3 and the corresponding AWT in Fig. 4, using $k = 5$. The UCT (an artificial example) has been optimized using the techniques discussed in Section IV-B, and the AWT has been optimized in three ways: We have removed states that are not reachable from the initial state, we have merged directions, and we have removed edges. (The last two optimizations are explained in the next subsections). Still, there is ample room for improvement of the AWT.

Application of Theorem 1 removes the 12 states below the dashed line on the bottom left and the incident edges. '

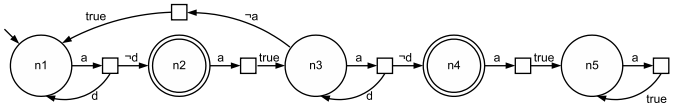


Fig. 3. UCT that requires rank 5. Edges that are not shown (for instance from n_4 with label $\neg a$) correspond to labels that are not allowed.

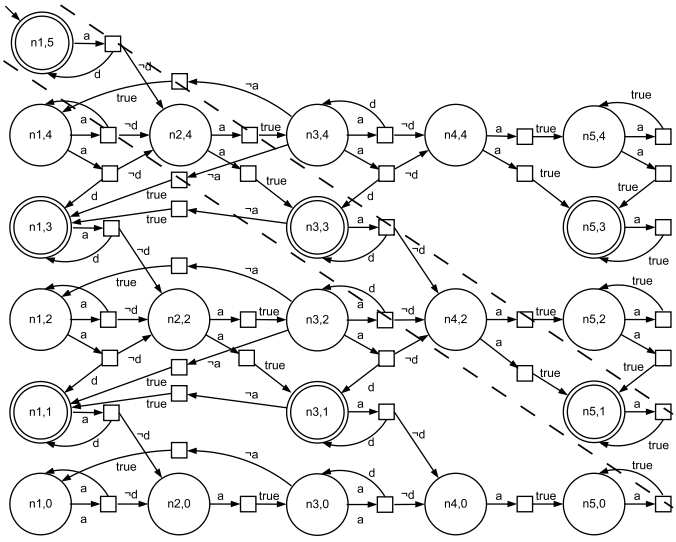


Fig. 4. AWT for UCT in Figure 3.

typical situation: each UCT state has an associated minimum rank. □

It should be noted that $A_{\text{AWT}k}$ has a layered structure: there are no states with rank j with a transition back to a state with a rank $i > j$. Furthermore, $A_{\text{AWT}k+1}$ consists of $A_{\text{AWT}k}$ plus one layer of states with rank $k+1$. This implies that game information computed for $A_{\text{AWT}k}$ can be reused for $A_{\text{AWT}k+1}$. A play is won (lost) in $A_{\text{AWT}k+1}$ if it reaches a states that is won (lost) in $A_{\text{AWT}k}$. Furthermore, if (q, j) is won, then so is (q, i) for $i > j$ when i is odd or j is even, which allows us to reuse some of the information computed for states with rank k when adding states with rank $k+1$. This follows from the fact that (q, i) simulates (q, j), as will be discussed in Subsection IV-C.3.

2) Merging Directions: Note that δ' may be drastically larger than δ: a single transition $C \in \delta(q, \sigma)$ yields $i^{|C|}$ transitions out of state $(q, i) \in Q'$. However, it turns out that it is not necessary to include conjuncts that send a copy to a (q, j) and (q, j') for $j \neq j'$. This is fortunate because it allows us to treat edges labeled with cubes over I as if they were labeled with directions.

Theorem 10: Let $A''_{\text{AWT}k} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ be as in Definition 8, but with

$$\delta''((q,i),\sigma) = \{C \in \delta'((q,i),\sigma) \mid \forall (d,(q,j)),(d',(q,j')) \in C, \text{ we have } j = j'\}.$$

We have $L(A''_{\text{AWT}k}) = L(A_{\text{AWT}k})$. □

Proof: Because $\delta''(q, \sigma) \subseteq \delta'(q, \sigma)$, any tree accepted by $A''_{\text{AWT}k}$ is also accepted by $A_{\text{AWT}k}$.

Let R be a run of $A_{\text{AWT}k}$, we will build a run R'' of $A''_{\text{AWT}k}$. Run R'' is isomorphic to R, using a bijection that maps node v of R to node v'' of R''. Run R'' has the same labels as R with the following exception. If node v in R is labeled $(t, (q, i))$ and has children $(t', (q', i'))$ and $(t'', (q', i''))$ with $i' > i''$, then the corresponding children of node v'' of R'' are labeled $(t', (q', i'))$ and $(t'', (q', i'))$.

Because in $A_{\text{AWT}k}$ state (q', i') has all transitions that (q', i'') has, R'' is a run of $A_{\text{AWT}k}$, and because it satisfies the extra condition on δ'' it is also a run of $A''_{\text{AWT}k}$. If R is accepting, then every infinite path π in R gets stuck in an odd rank w from some level l onwards. So starting from l, all children of nodes on π have rank at most w. That implies that the nodes on π in R'' have rank w starting at rank $l+1$ at the latest. Thus, π is still accepting, and since π is arbitrary, R'' is accepting as well. ■

This theorem is key to an efficient implementation as it allows us to represent a set of pairs $\{(d_1, q), \ldots, (d_k, q)\}$ as $(\{d_1, \ldots, d_k\}, q)$ whenever $\{d_1, \ldots, d_k\}$ can efficiently be represented by a cube over the input signals I.

3) Simulation minimization: We compute the simulation relation on $A_{\text{AWT}k}$ and use Theorems 3, 4, and 5 to optimize the automaton. We would like to point out one optimization in particular.

Lemma 11: For $(q, i), (q, j) \in Q'$ with $i \geq j$ such that i is odd or j is even, we have $(q, i) \succeq (q, j)$. □

Thus, for any σ, if i is even, we can remove all transitions $C \in \delta((q, i), \sigma)$ that include a pair (q', j) for $j \leq i - 2$. If i is odd we can additionally remove all transitions that contain a pair (q', j) with $q \notin \alpha$ and $j = i - 1$. That is, odd states become deterministic and for even states there are at most two alternatives to choose from.

Theorem 12: Let $A'_{\text{AWT}k} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ as in Definition 8, but with

$$\delta''((q,i),\sigma) = \big\{C \in \delta(q,\sigma) \mid \forall (d',(q',i')) \in C : i' \in \{i-1, i\}, (i \text{ is even} \vee q' \in \alpha \vee i' = i), \forall (d'',(q',i'')) \in C : i' = i''\}.$$

then $L(A'_{\text{AWT}k}) = L(A_{\text{AWT}k})$. □

Example 13: States $(n_4, 4)$, $(n_5, 4)$, and $(n_5, 3)$ (top right) are simulation equivalent with $(n_4, 2)$, $(n_5, 2)$, and $(n_5, 1)$, respectively. Using Theorem 3, we can remove states $(n_4, 4)$, $(n_5, 4)$, and $(n_5, 3)$, and redirect incoming edges to equivalent states.

Furthermore, the previous removal of the states on the bottom left implies that $(n_3, 4) \preceq (n_3, 3)$. Since $(n_2, 4)$ has identical transitions to $(n_3, 4)$ and $(n_3, 3)$, Theorem 5 allows us to remove the transition to $(n_3, 4)$. Thus, $(n_3, 4)$ becomes unreachable and can be removed. The same holds for $(n_5, 2)$ for a similar reason. (This optimization also allows us to remove states $(n_4, 4)$, $(n_5, 4)$, and $(n_5, 3)$, but Theorem 5 is not in general stronger than Theorem 3.)

The optimization of the edges due to Theorem 12 is already shown in Fig. 4. Consider, for instance, the transition from $(n_2, 4)$ to $(n_3, 4)$.

Altogether, we have reduced the number of states in the AWT from 22 to 5. The removal of edges is equally important as it reduces nondeterminism and makes the translation to an NBT more efficient.

D. NBT

The next step is to translate $A_{\text{AWT}k}$ to an NBT $A_{\text{NBT}k}$ with the same language. Assume that $A_{\text{AWT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. We first need some additional notation. For $S \subseteq Q$ and $\sigma \in \Sigma$ let

$$sat(S, \sigma) = \{C \in 2^{D \times Q} \mid C \text{ is a minimal set such that } \forall q \in S\, \exists C_q \in \delta(q, \sigma) : C_q \subseteq C\}.$$

For $(S, O) \in 2^Q \times 2^Q$, let

$$sat((S, O), \sigma) = \{(S', O') \in 2^Q \times 2^Q \mid S' \in sat(S, \sigma), O' \in sat(O, \sigma), O' \subseteq S'\}.$$

Furthermore, let $S_d = \{s \mid (d, s) \in S\}$, let $O_d = \{s \mid (d, s) \in O\}$. Let $C_N(S, O) = \{(d, (S_d, O_d \setminus \alpha)) \mid d \in D\}$ and let $C_\emptyset(S) = \{(d, (S_d, S_d \setminus \alpha)) \mid d \in D\}$.

Definition 14: [1], [16] Let $A_{\text{NBT}k} = (\Sigma, D, 2^Q \times 2^{Q \setminus \alpha}, (\{q_0\}, \emptyset), \delta', 2^Q \times \emptyset)$ with

$$\delta'((S, O), \sigma) = \begin{cases} \{C_N(S', O') \mid (S', O') \in sat((S, O), \sigma)\} & \text{if } O' \neq \emptyset \\ \{C_\emptyset(S') \mid S' \in sat(S, \sigma)\} & \text{otherwise} \end{cases}$$

□

We have $L(A_{\text{NBT}k}) = L(A_{\text{AWT}k})$.

We improve this construction in three ways. First, we make use of the simulation relation on the AWT to reduce the size of the NBT. Second, we remove *inconsistent states*, and third, we compute the NBT on the fly.

1) Simulation-Based Optimization: We can use the simulation relation that we have computed on $A_{\text{AWT}k}$ to approximate the simulation relation on $A_{\text{NBT}k}$. This is a simple extension of Fritz' result for word automata [17].

Given a direct simulation relation $\preceq_{\text{AWT}}$ for $A_{\text{AWT}k}$, we define the simulation relation $\preceq' \subseteq Q' \times Q'$ on $A_{\text{NBT}k}$ as

$$(S_1, O_1) \preceq' (S_2, O_2) \text{ iff}$$
$$\forall q_2 \in S_2\, \exists q_1 \in S_1 : q_1 \preceq_{\text{AWT}} q_2 \wedge (q_2 \in O_2 \rightarrow q_1 \in O_1).$$

Note that $\preceq'$ is a subset of the full (direct) simulation relation on $A_{\text{NBT}k}$ and thus, the following lemma holds.

Lemma 15: $(S_1, O_1) \preceq' (S_2, O_2)$ implies $L(A^{(S_1, O_1)}) \subseteq L(A^{(S_2, O_2)})$. □

In particular, for a state $(S, O) \in Q'$, if $q, q' \in S$, $q \preceq_{\text{AWT}} q'$, and $q' \in O \rightarrow q \in O$, then $(S, O) \simeq (S \setminus \{q'\}, O \setminus \{q'\})$. Thus, by Theorem 3, we can remove q' from such sets. Likewise, if $A_{\text{NBT}k}$ contains two simulation equivalent states (S, O) and (S', O') we keep only one (preferring the one with smaller cardinality). Finally, we can use Theorem 5 to remove states that have a simulating sibling.

2) Removing Inconsistent States: In [1], it is shown that it is not necessary to include states (S, O) such that (q, i) and $(q, j) \in S$ with $i \neq j$. This implies that we can use the following optimization.

Theorem 16: Let $A'_{\text{NBT}k} = (\Sigma, D, Q'', (\{q_0\}, \emptyset), \delta'', 2^Q \times \emptyset)$ be as in Definition 14, with $Q'' = Q \setminus \{(S, O) \mid \exists (q, i), (q, j) \in S : i \neq j\}$. The transition relation δ'' is obtained from δ' by replacing, for all $C \in \delta'(q, \sigma)$ and all $(S, O) \in C$, state (S, O) by (S', O') where S' is obtained from S by removing all states (q, j) with j not minimal and O' is obtained from O by replacing $(q, j) \in O$ by (q, j') if $(q, j) \notin S'$ and $(q, j) \in S'$.

We have $L(A'_{\text{NBT}k}) = L(A_{\text{NBT}})$. □

This is an important theorem as it reduces the number of states in the NBT to $(k+1)^{2n}$ instead of 2^{nk}, where n is the number of states in A_{UCT}.

3) On-the-Fly Computation: Suppose $A_{\text{NBT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. Instead of building $A_{\text{NBT}k}$ in full, we construct an NBT $A'_{\text{NBT}}[k] = (\Sigma, D, Q', q_0, \delta', \alpha \cap Q')$ such that $q_0 \in Q' \subseteq Q$ and for $q \in Q'$, either $\delta'(q, \sigma) = \delta(q, \sigma)$ for all σ or $\delta'(q, \sigma) = \emptyset$ for all σ. Thus, $L(A'_{\text{NBT}k}) \subseteq L(A_{\text{NBT}k})$. If $L(A'_{\text{NBT}k}) \neq \emptyset$, the witness of nonemptiness of $L(A'_{\text{NBT}k})$ is a witness of nonemptiness of $L(A_{\text{NBT}k})$. Otherwise, we select a state $q \in Q'$ with $\delta'(q, \sigma) = \emptyset$ and *expand* it, setting $\delta'(q, \sigma) = \delta(q, \sigma)$, introducing the necessary states to Q'.

Our current heuristic expands states in a breadth first manner, which is quite effective. It may be beneficial to expand certain state first, say states with a low cardinality or with high ranks.

E. Moore Machine

We use the game defined in Section III-A to compute language emptiness on the $A_{\text{NBT}k}$. Since $A_{\text{NBT}k}$ is nondeterministic, all states in the winning region have a nonempty language. If the initial state is in the winning region, the language of $A_{\text{NBT}k}$ is not empty and we extract a witness.

Since $A_{\text{NBT}k}$ is a subset of $A_{\text{NBT}k+1}$, we can reuse all results obtained when computing language emptiness on $A_{\text{NBT}k}$ to compute language emptiness on $A_{\text{NBT}k+1}$.

Moreover, it follows from Miyano and Hayashi's construction that if $L(A^{(S,O)}) \neq \emptyset$ and $S \subseteq S'$, then $L(A^{(S',O')}) \neq \emptyset$. We may use this fact to further speed up the computation of language emptiness, and especially to reuse information obtained computing language emptiness on $A_{\text{NBT}k}$ for larger k.

A witness for nonemptiness corresponds to a winning *attractor strategy* [18]. The winning strategy follows the μ-iterations of the final ν-computation of $W_B(\alpha)$: From a state $q \notin \alpha$ we go to a state q' from which the protagonist can force a shorter path to an accepting state. In an accepting state we move back to an arbitrary state in the winning region.

If a strategy exists, it corresponds to a complete Σ-labeled D-tree and thus to a Moore machine M. The states of M are the states of $A_{\text{NBT}k}$ that are reachable when the strategy is followed, and the edges are given by the strategy.

To minimize the strategy, we compute the simulation relation and apply Theorem 3, which is equivalent to using the classical FSM minimization algorithm [19]. Thus, the optimized strategy is guaranteed to be minimal with respect to its given I/O language. The output of our tool is a state machine described in Verilog that implements this strategy.

V. Experimental Results

Our tool, Lily[1], is an implementation of the synthesis algorithm and the optimizations described in this paper. It is

[1] http://www.ist.tugraz.at/staff/jobstmann/lily/

```
module tlc(hl,fl,clk,car,timer);
  input  clk,car,timer;
  output hl,fl;
  wire clk,hl,fl,car,timer;
  reg [0:0] state;

  assign hl = (state == 0);
  assign fl = (state == 1);

  initial begin
    state = 0;
  end
  always @(posedge clk) begin
    case(state)
    0: begin
      if (timer==0) state = 0;
      if (timer==1 && car==1) state = 1;
      if (car==0) state = 0;
    end
    1: begin
      if (timer==1) state = 0;
      if (timer==0) state = 1;
    end
    endcase
  end
endmodule //tlc
```

Fig. 5. Generated design for a simple traffic light

implemented on top of Wring [6], [20] and written in Perl. We have run our experiments on a Linux machine with a 2.8 GHz Pentium 4 CPU and 2 GB of RAM.

Example 17: Consider the following formula.

$$\begin{aligned}&\mathsf{G}\,\mathsf{F}\,timer \rightarrow \\ &\quad\big(\mathsf{G}(hl \rightarrow (hl\,\mathsf{U}\,timer)) \wedge \mathsf{G}(fl \rightarrow (fl\,\mathsf{U}\,timer)) \wedge \\ &\quad\mathsf{G}(\neg hl \vee \neg fl) \wedge \mathsf{G}(car \rightarrow \mathsf{F}\,\neg car \vee fl) \wedge \mathsf{G}\,\mathsf{F}\,hl \wedge \\ &\qquad\qquad\mathsf{G}(hl \rightarrow (hl\,\mathsf{W}\,car))\big).\end{aligned}$$

The formula specifies a small traffic light system consisting of two lights. The highway light is green iff $hl = \mathsf{true}$, and similarly for the crossing farm road and fl. Signals hl and fl form the output. The input signal car indicates that a car is waiting at the farm road and $timer$ represents the expiration of a timer. The specification assumes that the timer expires regularly. It requires that a green light stay green until the timer expires. Furthermore, one of the lights must always be red, every car at the farm road is eventually allowed to drive on (unless it disappears), the highway light is regularly set to green, and it does not become red until there is a car that wants to cross the highway. The specification is realizable and the design generated by Lily is shown in Figure 5. □

We show the effectiveness of the various optimizations by synthesizing 20 handwritten formulas, mostly different arbiters and some traffic light controllers. Our examples are small, but we show a significant improvement over the straightforward implementation.

For realizable formulas, we have verified the output of our tool with a model checker. We can verify that a formula is unrealizable by synthesizing an environment that forces the specification to be violated. Since a system is a Moore machine, an environment is a Mealy machine. Note that φ can be realized by a Mealy machine iff φ' can be realized by a Moore machine, where φ' is obtained from φ by replacing all occurences of an output o by $\mathsf{X}\,o$. This means that we can apply our approach to check that the environment is realizable.

We show our results in Table I. (The specification in example 17 has number 9.) In the column labeled *T,B,AP*, we provide the number of temporal operators, the number of Boolean operators, and the number of atomic propositions in the formula. We also give the strength in Column *strength* and the number of states and edges of the optimized NBW in Column *NBW* using the format states(edges).

The next six columns report time used with different combinations of optimizations. We write "> mem" if the run has exceeded the memory limit and "$> 3600s$" if it did not finish within one hour.

In Column *Plain* we give the time using only one optimization: transitions from a state with rank i go only to states with ranks $i-1$ and $i-2$, not to smaller ranks. Without this optimization, synthesis is impossible on most examples. Column *Plain+dm* shows the time used if we apply Theorem 10, which allows us to merge directions. In Column *UCT+dm*, we give the time usage of runs in which we applied game optimization (Theorem 1) on the UCT and we merged the directions. We show the results for applying all the optimizations suggested in Section IV-C on the AWT in Column *AWT+dm*. Column *UCT+dm* shows the time used if we apply the NBT optimizations and merge directions.

In the Column *All* we give the results for combining all optimizations. For realizable formulas, the number of states and edges of the design generated during those runs is given in the column labeled with *Witness*. We write n.r. if a formula is not realizable. The generated designs are minimized as described in Section IV-E.

In Column *NBT-all* we give the size (states(edges)) of the NBT using all optimizations. In contrast, in Column *NBT-plain+dm*, we show the size of the NBTs generated by the runs where we used only direction merging.

Our examples show that if the NBW for $\neg\varphi$ is strong and we have to construct the AWT, the straightforward implementation often fails to complete. (See Column *Plain*.) Five examples exceed the memory limit due to the expensive construction of the transition relation of the AWT. The optimizations due to Theorem 10, which allows us to merge directions, are necessary to overcome this limit. (See Column *Plain+dm*.)

If we optimize the UCT according to Theorem 1, we speed up about half of the examples by a factor of two or more. (Compare Column *Plain+dm* and *UCT+dm*, e.g., Line 2,3, or 9). The game based minimization is very effective if the formula or part of it is not realizable and is very cheap to compute. In particular for Examples 3 the difference is huge because the language of the optimized UCT is empty.

None of the optimizations on the AWT was very effective on their own. For example we still had two timeouts with game based optimization. The same holds for simulation based optimization. Further simplification of δ' according to Theorem 12 resulted in one timeout. Nevertheless, these optimizations are very efficient when combined as can be seen in Column *AWT+dm*.

TABLE I

No	*T,B,AP*	*Strength*	*NBW*	*Plain*	*Plain+dm*	*UCT+dm*	*AWT+dm*	*NBT+dm*	*All*	*NBT-plain+dm*	*NBT-all*	*Witness*
1	12,5,4	weak	8(14)	2.94 s	1.96 s	0.71 s	0.73 s	2.53 s	0.38 s	48(192)	0(0)	n. r.
2	6,3,4	strong	7(15)	> mem	1689.13 s	575.70 s	2.20 s	2.43 s	1.25 s	3943(973764)	6(28)	2(3)
3	4,4,4	strong	12(44)	> mem	> mem	3.73 s	>3600 s	> 3600 s	3.95 s	-	0(0)	n. r.
4	9,5,4	strong	6(11)	12.90 s	3.14 s	2.63 s	0.61 s	0.74 s	0.66 s	95(1104)	8(37)	6(11)
5	9,6,4	weak	7(19)	0.61 s	0.62 s	0.68 s	0.64 s	0.62 s	0.69 s	14(65)	14(65)	10(25)
6	15,12,4	weak	9(30)	3.43 s	2.94 s	3.01 s	2.97 s	3.26 s	3.15 s	58(502)	58(502)	43(145)
7	13,5,5	strong	7(15)	69.14 s	20.07 s	12.83 s	1.23 s	3.42 s	1.24 s	384(9564)	26(164)	15(31)
8	20,9,7	strong	9(22)	> 3600 s	258.81 s	294.29 s	6.41 s	13.32 s	5.98 s	1810(75264)	80(811)	41(109)
9	11,9,4	strong	9(19)	> mem	113.08 s	9.29 s	14.90 s	5.57 s	8.90 s	1079(81700)	101(840)	2(5)
10	12,5,4	weak	8(12)	3.64 s	1.28 s	0.62 s	0.70 s	1.31 s	0.35 s	28(116)	0(0)	n. r.
11	23,21,5	strong	24(90)	> mem	> mem	> 3600 s	17.57 s	94.29 s	15.91 s	-	31(167)	6(13)
12	40,24,8	weak	14(84)	201.18 s	219.38 s	58.82 s	61.02 s	37.95 s	32.41 s	251(88016)	26(456)	5(41)
13	10,9,4	strong	24(134)	> mem	> 3600 s	> 3600 s	522.21 s	> 3600 s	46.37 s	360(440093)	23(127)	17(75)
14	14,9,4	weak	13(34)	7.15 s	6.09 s	4.08 s	4.44 s	4.54 s	2.39 s	82(730)	21(92)	7(22)
15	16,10,10	weak	24(68)	90.63 s	68.49 s	13.60 s	13.10 s	17.00 s	8.29 s	618(8326)	27(142)	n. r.
16	16,13,4	weak	18(50)	8.65 s	6.33 s	6.45 s	7.23 s	7.71 s	3.93 s	142(1492)	17(112)	8(31)
17	19,13,4	weak	25(69)	24.52 s	21.88 s	11.28 s	12.97 s	14.46 s	8.18 s	368(3716)	25(162)	12(54)
18	16,17,4	weak	17(45)	11.14 s	9.14 s	4.58 s	5.51 s	6.05 s	4.14 s	118(730)	13(53)	8(23)
19	28,14,9	strong	11(30)	> mem	> mem	> 3600 s	72.78 s	483.38 s	39.26 s	755(83106)	242(3834)	124(444)
20	8,6,2	strong	7(14)	3.21 s	2.98 s	1.24 s	1.13 s	0.82 s	0.73 s	112(1187)	5(10)	2(3)

The simulation-based optimizations for the NBT (explained in Section IV-D) typically reduce the size of the resulting NBT between 60% and 90%. For example, in Example 9 the size of the NBT is reduced from about 1000 states to 300. For this example, the on-the-fly game computation further reduces the number of NBT-states to about 100. Only in the small examples is the entire state space of the NBT needed to compute a witness. (Cf. Column *NBT-plain+dm,NBT-all,Witness*).

In our examples, the UCT optimizations were crucial once to turn a strong UCT into a weak one (in this case with an empty language). In all other cases, they are outperformed by the AWT optimizations. The results of the NBT optimization are mixed: they can fail (Example 3 and 13) or perform better than any other optimization (Example 9 and 12). The combination of all optimizations is needed to finish all examples.

VI. Summary

The paper described the first implementation of a synthesis tool for full LTL. We have presented a set of optimizations for tree automata and have shown how these make a major difference in the efficiency of the implementation.

Further work is concerned with further increases in efficiency, debugging of specifications (especially of unrealizable ones), and with ways to effectively combine specifications with hand-written HDL code.

Acknowledgments: The authors would like to thank Orna Kupferman for stimulating discussions and patient explanations.

References

[1] O. Kupferman and M. Vardi, "Safraless decision procedures," in *Symposium on Foundations of Computer Science (FOCS'05)*, 2005, pp. 531–542.

[2] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. Symposium on Principles of Programming Languages (POPL '89)*, 1989, pp. 179–190.

[3] R. Rosner, "Modular synthesis of reactive systems," Ph.D. dissertation, Weizmann Institute of Science, 1992.

[4] S. Safra, "On the complexity of ω-automata," in *Symposium on Foundations of Computer Science*, Oct. 1988, pp. 319–327.

[5] M. Vardi, "A game-theoretic approach to automated program generation," 2005, Presentation at IFIP Working Group 2.11 Second Meeting. Available from http://www.cs.rice.edu/ taha/wg2.11/m-2/.

[6] F. Somenzi and R. Bloem, "Efficient Büchi automata from LTL formulae," in *Twelfth Conference on Computer Aided Verification (CAV'00)*, E. A. Emerson and A. P. Sistla, Eds. Berlin: Springer-Verlag, July 2000, pp. 248–263, LNCS 1855.

[7] N. Wallmeier, P. Hütten, and W. Thomas, "Symbolic synthesis of finite-state controllers for request-response specifications," in *Proceedings of the International Conference on the Implementation and Application of Automata*. Springer-Verlag, 2003.

[8] A. Harding, M. Ryan, and P. Schobbens, "A new algorithm for strategy synthesis in LTL games," in *Tools and Algorithms for the Construction and the Analysis of Systems (TACAS'05)*, 2005, pp. 477–492.

[9] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, 2006, pp. 364–380.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[11] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi, "Alternating refinement relations," in *Proc. 9th Conferance on Concurrency Theory*. Nice: Springer-Verlag, Sept. 1998, pp. 163–178, LNCS 1466.

[12] C. Fritz and T. Wilke, "State space reductions for alternating Büchi automata," in *Foundations of Software Technology and Theoretical Computer Science*. Kanpur, India: Springer-Verlag, Dec. 2002, pp. 157–168, LNCS 2556.

[13] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi, "On complementing nondeterministic Büchi automata," in *Correct Hardware Design and Verification Methods (CHARME'03)*. Berlin: Springer-Verlag, Oct. 2003, pp. 96–110, LNCS 2860.

[14] P. Wolper, M. Y. Vardi, and A. P. Sistla, "Reasoning about infinite computation paths," in *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, 1983, pp. 185–194.

[15] N. Piterman, "From nondeterministic Büchi and Streett automata to deterministic parity automata," in *21st Symposium on Logic in Computer Science (LICS'06)*, 2006, To appear.

[16] S. Miyano and T. Hayashi, "Alternating finite automata on ω-words," *Theoretical Computer Science*, vol. 32, pp. 321–330, 1984.

[17] C. Fritz, "Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata," in *Conference on Implementation and Application of Automata (CIAA'03)*, O. H. Ibarra and Z. Dang, Eds., 2003, pp. 35–48, LNCS 2759.

[18] W. Thomas, "On the synthesis of strategies in infinite games," in *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1995, pp. 1–13, LNCS 900.

[19] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.

[20] S. Gurumurthy, R. Bloem, and F. Somenzi, "Fair simulation minimization," in *Fourteenth Conference on Computer Aided Verification (CAV'02)*, E. Brinksma and K. G. Larsen, Eds. Berlin: Springer-Verlag, July 2002, pp. 610–623, LNCS 2404.

From PSL to NBA: a Modular Symbolic Encoding

Alessandro Cimatti
ITC-irst
cimatti@itc.it

Marco Roveri
ITC-irst
roveri@itc.it

Simone Semprini
ITC-irst
semprini@itc.it

Stefano Tonetta
University of Lugano
tonettas@lu.unisi.ch

***Abstract*— The IEEE standard Property Specification Language (PSL) allows to express all ω-regular properties mixing Linear Temporal Logic (LTL) with Sequential Extended Regular Expressions (SEREs), and is increasingly used in many phases of the hardware design cycle, from specification to verification.**

Many verification engines are able to manipulate Nondeterministic Büchi Automata (NBA), that can represent ω-regular properties. Thus, the ability to convert PSL into NBA is an important enabling factor for the reuse of a large wealth of verification tools.

Recent works propose a two-step conversion from PSL to NBA: first, the PSL property is encoded into an Alternating Büchi Automaton (ABA); then, the ABA is converted into an NBA with variants of Miyano-Hayashi's construction. These approaches are problematic in practice: in fact, they are often unable to carry out the conversion in acceptable time, even for PSL specifications of moderate size.

In this paper, we propose a modular encoding of PSL into symbolically represented NBA. We convert a PSL property into a normal form that separates the LTL and the SERE components. Each of these components can be processed separately, so that the NBA corresponding to the original PSL property is presented in the form of an implicit product, delaying composition until search time.

Our approach has two other advantages: first, we can leverage mature techniques for the LTL components; second, we leverage the particular form of the PSL components that appear in the normal form to improve over the general translation.

The transformation is proved correct. A thorough experimental analysis over large sets of paradigmatic properties (from patterns of properties commonly used in practice) shows that our approach drastically reduces the construction time of the symbolic NBA, and positively affects the overall verification time.

I. Introduction

The IEEE standard Property Specification Language PSL [1] is increasingly used as means to capture requirements on the behavior of a design, such as assumptions about the environment in which the design is expected to operate, internal behavioral requirements, and further constraints that arise during the design process from specification to verification.

The most important fragment of PSL combines Linear Temporal Logic (LTL) [2] with Sequential Extended Regular Expressions (SERE), a variant of classical regular expressions [1]. This combination results in ω-regular expressiveness, and enables to express many properties of practical interest in a compact and readable way.

Currently, most verification engines are able to manipulate Nondeterministic Büchi Automata (NBAs), thus allowing for the verification of all ω-regular properties. However, only few model checkers are able to deal with PSL, and those who do cover only a subset of the language. Thus, the ability to convert PSL into NBA would be an important enabling factor for the reuse of a large wealth of verification tools.

The translation of LTL augmented with regular expressions into NBA is exponential in the size of the input formula (see, e.g., [3]). In addition, SEREs extend regular expressions with the intersection operation, at the cost of an exponential blow-up. Thus, the translation of PSL may become a bottleneck even for real practical examples.

Following state-of-the-art techniques in LTL compilation [4], [5], recent approaches to PSL model checking [6], [7] tackle the translation into NBA in two steps. First, the PSL formula is encoded into an Alternating Büchi Automaton (ABA) [6]. The encoding builds a unique, monolithic ABA. Then, the monolithic ABA is converted into an NBA with variants of Miyano-Hayashi's construction (MH) [8] that, for an ABA of n states, generates an NBA of $O(3^n)$ states. In [9], a symbolic encoding of the NBA corresponding to the ABA of the PSL property is proposed. The authors of [7] exploit the fact that ABAs obtained from PSL are "weak" (see [10]) to directly define a symbolic encoding of the ABA. Both approaches try to limit the encoding size (delaying the explosion until search time). However, the explicit manipulation and optimization of the ABA is in practice a bottleneck: these approaches are often unable to carry out the conversion in acceptable time, even for PSL specifications of moderate size.

In this paper, we propose a modular direct encoding of PSL into a symbolically represented NBA without generating any explicit ABA. Our approach builds on the following main ideas.

First, we turn the PSL formula into a normal form, that we named *Suffix Operator Normal Form* (SONF). This normal form separates the SERE components and the LTL components. This makes the approach modular, i.e. rather than constructing a monolithic automaton we generate it in the form of an implicit product, thus delaying composition until search time. In addition, the two components can be encoded separately: the encoding of the LTL components can rely on mature techniques [4], [11], [12]; the PSL components can be encoded by any standard conversion to NBA. The resulting overall automaton is also more compact since it is the implicit synchronous composition of smaller automata.

Second, the interface between SERE and LTL is normalized, so that only specific PSL patterns, that we call *Suffix Operator Subformulas*, are possible. This allows us to define optimized encoding techniques specific to each of the patterns in order

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

to translate efficiently the Suffix Operator Subformulas into NBAs.

We prove that our transformation is correct, and evaluate our approach in the NUSMV model checker [13], with a thorough experimental analysis over large sets of PSL properties, generated from paradigmatic patterns of properties commonly used in practice [14]. This analysis shows that our approach dramatically reduces the construction time of the symbolic NBA. In addition, an evaluation on verification and language emptiness problems, using both BDDs and SAT, shows that our approach can positively affect the overall verification time.

A similar approach has been independently proposed in [15], where a PSL formula is translated into a tester, a finite-state machine that monitors if the suffix of the processed word satisfies the formula. The translation is bottom-up and compositional: each subformula is translated into an automaton which is then symbolically encoded using a boolean variable to monitor its satisfiability. The inductive step is very similar to our basic rule for SONF reduction. However, we do not build an automaton for every subformula, but we simply separate the LTL part from the SERE part and we leave the freedom to use different translation for each part; moreover, we use different optimized compilation for the suffix conjunction and the suffix implication.

The paper is structured as follows. In Section II we present the syntax and semantics of PSL, and some background on automata. In Section III we overview the monolithic approach, and discuss its inadequacies. In Section IV we define the Suffix Operator Normal form, and in Section V we discuss the encoding, and we show how to exploit the structure of the SONF to generate NBAs. In Sections V-B.1 and V-B.2, we describe the optimized translations for Suffix Operator Subformulas. In Section VI we experimentally evaluate our approach. Finally, in Section VII we draw some conclusions and discuss directions for future research.

The proofs of the main theorems and additional experimental results are reported in [16].

II. Technical Background

A. The property specification language PSL

PSL is a very rich language [1]. We consider the subset of PSL that combines Linear Temporal Logic [2] (LTL) and Sequential Extended Regular Expressions (SERE), a variant of classical regular expressions [1]. This subset provides ω-regular expressiveness, it is the mostly used in practice and constitutes *the core* of the PSL temporal layer [1]. We will not deal with PSL "clocked" expression that are not part of the core since any clocked expression can be rewritten into an equivalent un-clocked one. The same applies for the PSL "abort" operator that can be efficiently rewritten into pure LTL as shown in [17].

In the following, we assume as given a set $\mathcal{AP}$ of atomic propositions that intuitively represent the states of the system under verification. Let $\Sigma := 2^{\mathcal{AP}}$ be the alphabet of the languages defined by the system and the properties. Let $\mathcal{A} \supseteq \mathcal{AP}$ be a superset of atomic formulas[1]. Let $\Sigma_{\mathcal{A}} := 2^{\mathcal{A}}$ be the alphabet related to $\mathcal{A}$. Given a generic set V, we denote with $\mathcal{B}^+(V)$ the set of Boolean formulas obtained by applying only disjunction ($\vee$) and conjunction ($\wedge$) to elements in $V \cup \{\text{true}, \text{false}\}$; with $\mathcal{B}^\vee(V)$ the set of Boolean formulas obtained by applying only disjunction $\vee$ to elements in $V \cup \{\text{true}, \text{false}\}$; and with $\mathcal{B}^\neg(V)$ the set of Boolean formulas obtained by applying disjunction, conjunction and negation ($\neg$) to elements in $V \cup \{\text{true}, \text{false}\}$.

We denote a letter from a given alphabet Σ by ℓ, a word from Σ by v or w, and the concatenation of v and w by vw. We denote with $|w|$ the length of word w. A finite word $w = \ell_0 \ell_1 \dots \ell_n$ has length $n+1$, an infinite word has length ω. For $i < |w|$ we use w^i to denote the $(i+1)^{th}$ letter of w, and we denote with $w^{i\cdot\cdot}$ the suffix of w starting at w^i. When $i \leq j \leq |w|$, we denote with $w^{i..j}$ the finite sequence of letters starting from w^i and ending in w^j: $w^{i..j} := w^i w^{i+1} \dots w^j$.

SEREs are the PSL version of regular expressions. In particular, they extend the standard regular expressions with language intersection. This allows for a greater succinctness, but it implies a possible exponential blow-up in the conversion to automata. Formally,

*Definition 1 (*SERE*s syntax):*

- if b is Boolean expression, then b is a SERE;
- if r is a SERE, then r**[*]** is a SERE;
- if r_1 and r_2 are SEREs, then the following are SEREs

$$r_1 \texttt{ ; } r_2 \qquad r_1 \texttt{ : } r_2 \qquad r_1 \texttt{ | } r_2$$
$$r_1 \texttt{ \& } r_2 \qquad r_1 \texttt{ \&\& } r_2$$

SEREs can be concatenated with the operators **;** and **:**, the former for the consecutive concatenation of two sequences, the latter for one-state overlapping concatenation. The conjunction operators **&** and **&&** can be used to specify overlapping sequences, the latter for length-matching sequences. Disjunction can be specified using the **|** operator. The **[*]** operator specifies finite consecutive repetitions.

The semantics of SEREs is formally defined over *finite* words using, as the base case the semantics of Boolean expressions over letters in Σ, denoted with $\models_B$ hereafter.

*Definition 2 (*SERE*s semantics):* Given a Boolean expression b, a SERE r, and a *finite word* w, we define the satisfaction relation $w \models r$ as follows:

- $w \models b$ iff $|w| = 1$ and $w^0 \models_B b$
- $w \models r_1 \texttt{ ; } r_2$ iff $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1$, $w_2 \models r_2$
- $w \models r_1 \texttt{ : } r_2$ iff $\exists w_1, w_2, \ell$ s.t. $w = w_1 \ell w_2$, $w_1 \ell \models r_1$, $\ell w_2 \models r_2$
- $w \models r_1 \texttt{ | } r_2$ iff $w \models r_1$ or $w \models r_2$
- $w \models r_1 \texttt{ \& } r_2$ iff
  - $w \models r_1$ and $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_2$, or
  - $w \models r_2$ and $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1$
- $w \models r_1 \texttt{ \&\& } r_2$ iff $\exists w_1, w_2$ s.t. $w = w_1 w_2$, $w_1 \models r_1$, $w_1 \models r_2$

[1] We rewrite formulas by introducing new atoms. You can think of these fresh atoms as the existentially quantified propositions of QLTL [18]. Thus, their role is similar to the labels in standard CNF-ization of SAT instances.

- $w \models r$**[*]** iff $|w| = 0$ or $\exists w_1, w_2$ s.t. $|w_1| \neq 0$, $w = w_1 w_2$, $w_1 \models r$, $w_2 \models r$**[*]**

In the definition of the PSL syntax, for technical reasons, we introduce the "releases" operator (that is the dual of the "until" operator), and also we introduce the "suffix conjunction" connective as a dual of the suffix implication. Moreover, we consider only the strong version of all the operators (the weak operators can be rewritten in terms of the strong ones [1]).

*Definition 3 (*PSL *syntax):* We define the PSL formulas over $\mathcal{A}$, as follows:

- if $p \in \mathcal{A}$, p is a PSL formula;
- if ϕ_1 and ϕ_2 are PSL formulas, then $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ are PSL formulas;
- if ϕ_1 and ϕ_2 are PSL formulas, then $\mathbf{X}\,\phi_1$, $\phi_1\,\mathbf{U}\,\phi_2$, $\phi_1\,\mathbf{R}\,\phi_2$ are PSL formulas;
- if r is a SERE and ϕ is a PSL formulas, then $r \diamond\!\!\rightarrow \phi$ and $r \mapsto \phi$ are PSL formulas;
- if r is a SERE, then r is a PSL formula.

The **X** ("next-time"), the **U** ("until"), and the **R** ("releases") operators are called *temporal operators*. We call the $\diamond\!\!\rightarrow$ ("suffix conjunction"), and the $\mapsto$ ("suffix implication"), *suffix operators*. Notice that, the r not occurring in the left side of a suffix operator is the *strong* version of a SERE ($r!$ in the PSL notation). In the following, we will consider such r as an abbreviation for $r \diamond\!\!\rightarrow$ true [1], [6]. We also use $\mathbf{G}\,\phi$ as an abbreviation for false $\mathbf{R}\,\phi$. LTL can be seen as a subset of PSL in which the suffix operators and the SEREs are suppressed.

We interpret PSL expressions over *infinite* words, and we consider only the *strong semantics* of PSL [1] (though our approach can be easily extended to deal also with the weak semantics).

*Definition 4 (*PSL *semantics):* Let $w \in \Sigma^\omega$.

- $w \models p$ iff $w^0 \models_B p$;
- $w \models \neg\phi$ iff $w \not\models \phi$;
- $w \models \phi \wedge \psi$ iff $w \models \phi$ and $w \models \psi$;
- $w \models \phi \vee \psi$ iff either $w \models \phi$ or $w \models \psi$;
- $w \models \mathbf{X}\,\phi$ iff $|w| > 1$ and $w^{1\cdot\cdot} \models \phi$;
- $w \models \phi\,\mathbf{U}\,\psi$ iff, for some $j \geq 0$, $w^{j\cdot\cdot} \models \psi$ and, for all $0 \leq k < j$, $w^{k\cdot\cdot} \models \phi$;
- $w \models \phi\,\mathbf{R}\,\psi$ iff, for all $j \geq 0$, either $w^{j\cdot\cdot} \models \psi$ or, for some $0 \leq k < j$, $w^{k\cdot\cdot} \models \phi$;
- $w \models r \diamond\!\!\rightarrow \phi$ iff, for some $j \geq 0$, $w^{0..j} \models r$ and $w^{j\cdot\cdot} \models \phi$;
- $w \models r \mapsto \phi$ iff, for all $j \geq 0$, if $w^{0..j} \models r$, then $w^{j\cdot\cdot} \models \phi$.

Notice that we can build Boolean expressions by means of atomic formulas and Boolean connectives.

Definition 5 (language of PSL *formulas):* The language of a PSL formula ϕ over the alphabet $\Sigma_\mathcal{A}$ is defined as follows:

$$\mathcal{L}_\mathcal{A}(\phi) := \{w \in \Sigma_\mathcal{A}^\omega \mid w \models \phi\}$$

The language accepted by ϕ is defined as follows:

$$\begin{aligned}\mathcal{L}(\phi) \quad := \quad & \{w \in \Sigma^\omega \mid \text{for some } v \in \mathcal{L}_\mathcal{A}(\phi),\\ & v^i \cap \mathcal{AP} = w^i \text{ for all } i \geq 0\}\end{aligned}$$

B. Automata

Definition 6 (NFA): A Non-deterministic Finite-state Automaton (NFA) is a tuple $A = \langle \mathcal{A}, Q, q_0, \rho, F\rangle$, where

- $\mathcal{A}$ is the set of atoms;
- Q is a set of states;
- $q_0 \in Q$ is the initial state;
- $\rho : Q \times \Sigma_\mathcal{A} \rightarrow 2^Q$ is the transition function;
- $F \subseteq Q$ is the set of final states.

A *run* of an NFA A over the finite word $w = \ell_0, \ell_1, ..., \ell_n \in \Sigma_\mathcal{A}^*$ is a finite sequence of states $\pi = q_0, q_1, ..., q_n$ such that, for $0 \leq i < n$, $q_{i+1} \in \rho(q_i, \ell_i)$, and $q_n \in F$.

Definition 7 (NFA language): The language $\mathcal{L}_\mathcal{A}(A)$ of an NFA A related to the alphabet $\Sigma_\mathcal{A}$ is the set of words w such that there exists a run of A over w.

$$\begin{aligned}\mathcal{L}(A) \quad := \quad & \{w \in \Sigma^* \mid \text{for some } v \in \mathcal{L}_\mathcal{A}(A),\\ & v^i \cap \mathcal{AP} = w^i \text{ for all } i \geq 0\}\end{aligned}$$

Definition 8 (DFA): A Deterministic Finite-state Automaton (DFA) is an NFA $A = \langle \mathcal{A}, Q, q_0, \rho, F\rangle$ such that, for all $q \in Q$ and $\ell \in \Sigma_\mathcal{A}$, $|\rho(q, \ell)| \leq 1$.

Definition 9: A (Non-)Deterministic Finite-state Automaton is *complete* iff for all $q \in Q$, for all $\ell \in \Sigma_\mathcal{A}$, $|\rho(q, \ell)| \geq 1$.

SERE, NFA and DFA have the same expressive power. In particular, the following theorems are relevant.

Theorem 1: Fer every SERE r, there exists a complete NFA A_r such that $\mathcal{L}(A_r) = \mathcal{L}(r)$ and $|A_r| = |r|$ if r does not contain the **&&** operator, $|A_r| = 2^{O(|r|)}$ otherwise.

Theorem 2: Given an NFA A, we can build a DFA A' such that $\mathcal{L}(A') = \mathcal{L}(A)$ and $|A'| = 2^{O(|A|)}$.

Definition 10 (ABA): An Alternating Büchi Automaton (ABA) over infinite words is a tuple $A = \langle \mathcal{A}, Q, Q_0, \rho, F\rangle$, where

- $\mathcal{A}$ is the set of atoms;
- Q is a finite nonempty set of states;
- $Q_0 \subseteq Q$ is the set of initial states;
- $\rho : Q \times \Sigma_\mathcal{A} \rightarrow \mathcal{B}^+(Q)$ is the transition function;
- $F \subseteq Q$ is a set of accepting states.

A *run* of an ABA on an infinite word w is a (possibly infinite) Q-labeled tree $\tau = (\mathcal{T}, L)$ such that $L(\epsilon) \in Q_0$ and for every node $t \in \mathcal{T}$, t has at most $|Q|$ children and, if t is at the i-th level of τ, $L(t) = q$, and the children of t are $t_1, ..., t_k$, then $L(t_1), ..., L(t_k)$ satisfy $\rho(q, w^i)$. A run tree τ is *accepting* if every branch has infinite depth and features infinitely many labels in F.

Definition 11 (NBA): A Non-deterministic Büchi Automaton (NBA) is the tuple $A = \langle \mathcal{A}, Q, Q_0, \rho, F\rangle$, where

- $\mathcal{A}$ is the set of atoms building the labels;
- Q is a set of states;
- $Q_0 \subseteq Q$ is the set of initial states;
- $\rho : Q \times \Sigma_\mathcal{A} \rightarrow 2^Q$ is the transition function;
- $F \subseteq Q$ is the set of accepting states.

A *run* of an NBA A over the infinite word $w = \ell_0, \ell_1, ... \in \Sigma_\mathcal{A}^\omega$ is an infinite sequence of states $\pi = q_0, q_1, ...$ starting from some $q_0 \in Q_0$ such that, for all $i \geq 0$, $q_{i+1} \in \rho(q_i, \ell_i)$, and $q_i \in F$ for infinitely many i.

The runs of an NBA can be seen as trees with a single branch. Hence the accepting condition collapses to require that a run is accepting if it features infinitely many occurrences of states in F.

Definition 12 (ABA and NBA language): The language $\mathcal{L}_{\mathcal{A}}(A)$ of an ABA or NBA A related to the alphabet $\Sigma_{\mathcal{A}}$ is the set of words w such that there exists a run of A over w. The language of A is defined as follows:

$$\mathcal{L}(A) := \{w \in \Sigma^{\omega} \mid \text{for some } v \in \mathcal{L}_{\mathcal{A}}(A), \; v^i \cap \mathcal{AP} = w^i \text{ for all } i \geq 0\}$$

For any Alternating Büchi Automaton A, there exists a Nondeterministic Büchi Automaton B accepting the same language. Given an ABA A, the algorithm of [8] produces an NBA B such that with the same language. In the following, we rely on a simplified version [19]:

Theorem 3 (From ABA to NBA): For any ABA A there exists an NBA B such that $\mathcal{L}(B) = \mathcal{L}(A)$. Given $A = \langle \mathcal{A}, Q, Q_0, \rho, F \rangle$, B is defined as $B = \langle \mathcal{A}, Q_B, Q_{0B}, \rho_B, F_B \rangle$, where:

- $Q_B = \{(L, R) | L \in 2^Q, R \in 2^{Q \setminus F}\}$
- $Q_{0B} = Q_0 \times \emptyset$
- if $R \neq \emptyset$, then $\rho_B((L,R),\ell) = \{(L', R' \setminus F) \mid L' \models \bigcap_{q \in L} \rho(q,\ell), R' \subseteq L', R' \models \bigcap_{q \in R} \rho(q,\ell)\}$
  if $R = \emptyset$, then $\rho_B((L,R),\ell) = \{(L', L' \setminus F) \mid L' \models \bigcap_{q \in L} \rho(q,\ell)\}$
- $F_B = 2^Q \times \{\emptyset\}$

C. Fair Transition Systems

In the following, in order to simplify the presentation, we introduce the notion of *Fair Transition System* [20].

Definition 13: A *fair transition system (FTS)* is a tuple $S = \langle V, \mathcal{A}, T, I, F \rangle$, where V is a finite set of *state variables*, $\mathcal{A}$ is a finite set of *input variables*, $T \in \mathcal{B}^{\neg}(V \cup \mathcal{A} \cup V')$ is the *transition relation* (V' is the set of primed versions of variables in V), $I \in \mathcal{B}^{\neg}(V)$ specifies the set of *initial states*, and $F \in \mathcal{B}^{\neg}(V)$ specifies the *acceptance condition*.

An FTS $S = \langle V_s, \mathcal{A}, T_s, I_s, F_s \rangle$ defines an NBA A as follows: $A = \langle \mathcal{A}, 2^{V_s}, Q_0, \rho, F \rangle$, where Q_0 is defined by I_s, $\rho(q,\ell) = \{q' \mid (q, \ell, q') \models T\}$, and $F = \{q \mid q \models F_s\}$. Thus, we can speak of a run of an FTS and the language of an FTS as if it were an NBA.

Using FTS instead of NBA further stresses the fact that our encoding is symbolic and does not imply the construction of any explicit NBA.

III. Monolithic Encoding of PSL into NBA

In this section, we recall the state-of-the-art techniques to compile PSL formulas into automata. More details can be found in [6].

The translation proceeds bottom-up: it builds the automata corresponding to the SEREs and combine them with the temporal and the suffix operators. The resulting automaton is an ABA which is then translated into an NBA by the Miyano-Hayashi (MH) algorithm [8].

Thus, the basic step of the overall translation is the compilation of SEREs into NFAs. Concatenation (**;** , **:**) and union (**|**) are translated linearly in the standard way [21]. Instead, the intersection operators (**&** , **&&**) allow for a greater succinctness but imply a potential blow-up in the translation. The complexity stems from the difficulty of codifying the fact that two automata must synchronize on the end of the words that are accepted [22], [23]. [6] takes the conjunction of the two automata and removes the subsequent alternation by means of a subset construction.

Pure LTL formulas can be translated easily into a fragment of ABA, called linear weak ABA [4], [24]. This translation can be extended to handle automata as leaves of the LTL formulas, instead of the propositional atoms, yielding a general ABA.

Finally, the suffix operators are handled so that the NFA of the left-side SERE is combined with the ABA of the right-side PSL formula. In the case of a suffix implication, we require that every finite prefix satisfying the SERE is followed by a suffix satisfying the PSL formula; this requires a determinization and completion of the NFA. In the case of a suffix conjunction, there must exist a finite prefix satisfying the SERE followed by a suffix satisfying the PSL formula; this requires a fairness condition for the fulfillment of the prefix.

The bottom-up construction ends up into a monolithic ABA that encodes the language of the input PSL formula. The alternation is then removed by means of an exponential subset construction. A further quadratic blow up is necessary to handle the fairness condition [8]. It is possible to use a symbolic version of MH [9], so that the potential explosion is delayed at run time.

Remark 1: We can identify two main approaches to LTL compilation into NBA: on one hand, *syntactic compilers* (such as `ltl2smv` [25]) translate the LTL formula directly into a linear size FTS by introducing one variable for each subformula; on the other hand, *semantic compilers* (such as `Wring` [11]) translate the formula into an intermediate explicit representation, which is optimized and then symbolically represented by means of a logarithmic encoding. [12] showed that the latter approach usually performs better in terms of verification time. Its main drawback is that the automata optimization are often so expensive that the compilation time may result in a bottleneck. The monolithic approach described in this Section extends the semantic compilation of LTL preventing the use of different encodings for the different classes of subformulas. Moreover, it implies an explicit automaton construction that one may wish to avoid. As a further point, the classic syntactic compilation of [25] can be seen as a linear encoding of the corresponding alternating automaton. The main difference with MH stands in the fairness manipulation: if the input formula contains n **U** -subformulas, the classic construction produces n fairness constraints that result in n inner fixpoint computations at search time; on the opposite, MH produces only one fairness constraint but it must introduce n symbolic variables to keep track of which **U** -subformulas must be fulfilled. We are not aware of any comparison between the two approaches.

IV. Suffix Operator Normal Form for PSL

In this section we define the Suffix Operator Normal Form (SONF) for PSL. The first step is to extend the Negative Normal Form (standard for LTL) to the case of PSL.

Definition 14 (NNF): A PSL formula is in negated normal form (NNF) iff all the negations occur only in front of propositions.

Lemma 1 (NNF-ization): A PSL formula ϕ can always be reduced to an equivalent NNF $\mathcal{N}(\phi)$.

Proof: Each of the following transformations preserves the language.

- $\mathcal{N}(\neg p) := \neg p$,
- $\mathcal{N}(\neg(\phi_1 \vee \phi_2)) := \mathcal{N}(\neg\phi_1) \wedge \mathcal{N}(\neg\phi_2)$,
- $\mathcal{N}(\neg(\phi_1 \wedge \phi_2)) := \mathcal{N}(\neg\phi_1) \vee \mathcal{N}(\neg\phi_2)$,
- $\mathcal{N}(\neg(\phi_1 \mathbf{U} \phi_2)) := \mathcal{N}(\neg\phi_1) \mathbf{R} \mathcal{N}(\neg\phi_2)$,
- $\mathcal{N}(\neg(\phi_1 \mathbf{R} \phi_2)) := \mathcal{N}(\neg\phi_1) \mathbf{U} \mathcal{N}(\neg\phi_2)$,
- $\mathcal{N}(\neg(r \diamond\!\!\rightarrow \phi_1)) := r \mapsto \mathcal{N}(\neg\phi_1)$,
- $\mathcal{N}(\neg(r \mapsto \phi_1)) := r \diamond\!\!\rightarrow \mathcal{N}(\neg\phi_1)$.

■

We now provide a set of rewriting rules to convert the formula in a normal form named Suffix Operator Normal Form (SONF). Intuitively, a formula in SONF is structured as follows

$$\overbrace{\bigwedge_i \phi_i}^{\Psi_{LTL}} \wedge \overbrace{\bigwedge_j \mathbf{G}\,(p_I^j \rightarrow (r_j \star\!\!\rightarrow p_F^j))}^{\Psi_{PSL}}$$

where ϕ_i are LTL formulas, r_j are SEREs, and p_I^j and p_F^j are atoms.

Given a formula ϕ, the rewriting rules build ϕ', adding new atoms while preserving the language so that a model of ϕ' restricted to the original set of atomic propositions is a model of ϕ. For every subformula of ϕ of the form $r \diamond\!\!\rightarrow \psi$ (resp., $r \mapsto \psi$), we introduce two new atoms: $P_{r \diamond\rightarrow \psi}$ (resp., $P_{r \mapsto \psi}$) and P_ψ. We define the following rewriting rules:

Definition 15:

$$\begin{aligned} \phi[r \diamond\!\!\rightarrow \psi] \quad \Rightarrow \quad & \phi[P_{r \diamond\rightarrow \psi}] \wedge \\ & \mathbf{G}\,(P_{r \diamond\rightarrow \psi} \rightarrow (r \diamond\!\!\rightarrow P_\psi)) \wedge \\ & \mathbf{G}\,(P_\psi \rightarrow \psi) \end{aligned} \tag{1}$$

$$\begin{aligned} \phi[r \mapsto \psi] \quad \Rightarrow \quad & \phi[P_{r \mapsto \psi}] \wedge \\ & \mathbf{G}\,(P_{r \mapsto \psi} \rightarrow (r \mapsto P_\psi)) \wedge \\ & \mathbf{G}\,(P_\psi \rightarrow \psi) \end{aligned} \tag{2}$$

where $\phi[\psi] \Rightarrow \phi[P]$ means that we substitute every occurrence of ψ in ϕ with P. Intuitively, we substitute the suffix operator $r \star\!\!\rightarrow \psi$ with the corresponding activation predicate $P_{r \star\rightarrow \psi}$, and we add two global formulas at top-level: the first states that $P_{r \star\rightarrow \psi}$ always triggers $r \star\!\!\rightarrow P_\psi$; the second states that P_ψ always triggers ψ.

Example 1: The SONF of $\mathbf{F}\,\mathbf{G}\,(a\mathbf{[*]}\,\mathbf{;}\; b \mapsto \mathbf{F}\,c)$ is $\mathbf{F}\,\mathbf{G}\,(p_1) \wedge \mathbf{G}\,(p_1 \rightarrow a\mathbf{[*]}\,\mathbf{;}\, b \mapsto p_2) \wedge \mathbf{G}\,(p_2 \rightarrow \mathbf{F}\,c)$.

Notice that $|\phi'| = O(|\phi|)$. It is easy to see that the rewriting procedure terminates and always produces the same formula (modulo renaming). The following theorem guarantees the correctness of the rewriting rules.

Theorem 4: Let ϕ be a PSL formula over $\mathcal{A}$ and ψ a PSL subformula of ϕ that occurs only positively in ϕ. If

$$\phi' := \phi[P/\psi] \wedge \mathbf{G}\,(P \rightarrow \psi).$$

then $\mathcal{L}(\phi) = \mathcal{L}(\phi')$.

V. A Modular Translation from PSL to NBA

A. The Overall Approach

Algorithm 1 depicts the pseudo code of the `ModPsl2Ba` procedure for translating PSL to NBA in a modular manner. `ModPsl2Ba` relies on two building blocks, `Psl2Ba` and `Ltl2Ba`. `Psl2Ba` is the procedure described in Section III, that builds an automaton from a PSL formula; `Ltl2Ba` is a procedure that builds an NBA automaton from an LTL formula (for instance [25]).

The `ModPsl2Ba` procedure improves over `Psl2Ba` by transforming the PSL formula into an equivalent one in SONF. The `Sonf` procedure generates the SONF of a formula ϕ, thus decomposing it into subformulas according to their nature.This normalization is then exploited in two ways: first, we keep the resulting NBA partitioned (rather than monolithic); second, we call the tableau constructor `Ltl2Ba`, that is optimized for LTL, on the LTL part.

```
ModPsl2Ba(φ)
input  : φ the PSL input formula
output : a set S of NBAs;
         the final NBA is the product of all NBAs in S
begin
    φ' := Sonf(φ);
    S := ∅;
    /* φ' is in the form Ψ_LTL ∧ Ψ_PSL          */
    for ψ ∈ Ψ_LTL do
        A := Ltl2Ba(ψ);
        S := S ∪ {A};
    end
    for ψ ∈ Ψ_PSL do
        A := Psl2Ba(ψ);
        S := S ∪ {A};
    end
    return S
end
```

Algorithm 1: Modular translation.

Intuitively, the approach can be seen as a way to decompose the property in small pieces, apply to each piece the most effective encoding, and then gluing together the results by synchronous composition.

We remark that the result of the translation is a set of implicitly synchronized NBAs, while the approach described in III returns a single NBA. This enables a greater efficiency since we can exploit standard techniques of conjunctive partitioning in model checking [26]. It also allows multiple fairness conditions, which may improve the verification time by decomposing the fixpoint computation in smaller ones.

B. Encoding Suffix Operator Subformulas

Algorithm 1 can be further optimized. In fact, we notice that the expressions in Ψ_{PSL} contain exactly two specific patterns: namely, $\mathbf{G}\,(P_I \rightarrow (r \diamond\!\!\rightarrow P_F))$ and $\mathbf{G}\,(P_I \rightarrow (r \mapsto P_F))$. We call these kind of subformulas, *Suffix Operator Subformulas*. Here we present an optimized version of `Psl2Ba`, that works only for the Suffix Operator Subformulas, and builds the NBA without passing through the construction of the ABA. As a consequence of the optimization, the overall

algorithm ModPsl2Ba becomes independent of any ABA representation.

We exploit the structure of the Suffix Operator Subformulas that have a fixed pattern parametrized only on the SERE r. This way, given the NFA $A_r = \langle \mathcal{A}, Q, q_0, \rho, F\rangle$ of r, we can obtain directly the NBA A_ϕ corresponding to the Suffix Operator Subformula ϕ. In the following two sections, we will describe the symbolic encoding of such NBA as an FTS which is produced on top of the symbolic encoding of A_r. Notice that the encoding is linear in the size of A_r and we do not add new variables either for the top-level $\mathbf{G}$ operator or for the propositions P_I and P_F.

1) Encoding $\phi := \mathbf{G}\,(P_I \rightarrow (r \mapsto P_F))$ into NBA A_ϕ: The NFA A_r is first completed and determinized. The cost of these operations is alleviated by considering a symbolic representation of the labels in the NFA. Instead of representing the transitions by means of a function $\rho : Q \times \Sigma_{\mathcal{A}} \rightarrow 2^Q$ depending on the letters of the alphabet, we use a function $\rho : Q \rightarrow 2^{\mathcal{B}^{\neg}(\mathcal{A})\times Q}$ depending on Boolean combinations of atomic formulas. Then, we can obtain a symbolic representation of the deterministic and completed version of A_r by introducing a set of variables $V := \{v_q\}_{q\in Q}$ and a triple of formulas I_r, T_r, F_r where:

- $I_r := v_{q_0}$,
- $T_r := \bigwedge_{q\in Q}(v_q \rightarrow (\bigvee_{C\subseteq\rho(q)}(\bigwedge_{(a,q')\in C}(a \wedge v'_{q'}) \wedge \bigwedge_{(a,q')\in\rho(q)\setminus C} \neg a)))$,
- $F_r := \bigvee_{q\in F} v_q$.

Notice that, in $C \subseteq \rho(q)$ we consider also the empty set $\emptyset$.

The FTS S_ϕ is defined as $\langle V_\phi, \mathcal{A}, T_\phi, I_\phi, F_\phi\rangle$, where $V_\phi = V$, $I_\phi := \top$, $F_\phi := \top$, and

$$T_\phi := P_I \rightarrow I_r \wedge T_r[v'_q \wedge P_F / v'_q]_{q\in F}$$

.

By Definition 13, we can build an NBA A_ϕ from S_ϕ, such that $\mathcal{L}(A_\phi) = \mathcal{L}(S_\phi)$, and by the following theorem we can conclude $\mathcal{L}(S_\phi) = \mathcal{L}(\phi)$.

Theorem 5: $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$.

Example 2: Consider the suffix implication subformula $\mathbf{G}\,(p_1 \rightarrow a[\!*\!]\,;\, b \mapsto p_2)$ of Example 1. Suppose the NFA related to $r = a[\!*\!]\,;\, b$ is

$$A_r = \langle\{q_1, q_2\}, q_1, \{(q_1, a, q_1), (q_1, b, q_2)\}, \{q_2\}\rangle.$$

Then the transition relation of the final FTS is:

$$\begin{array}{ll}(p_1 \rightarrow v_{q_1}) \wedge (v_{q_1} \rightarrow & ((a \wedge b \wedge v'_{q_1} \wedge v'_{q_2} \wedge p_2) \vee \\ & (a \wedge \neg b \wedge v'_{q_1}) \vee \\ & (\neg a \wedge b \wedge v'_{q_2} \wedge p_2) \vee \\ & (\neg a \wedge \neg b))).\end{array}$$

2) Encoding $\phi := \mathbf{G}\,(P_I \rightarrow (r \Diamond\!\!\rightarrow P_F))$ into NBA A_ϕ: We first obtain a symbolic representation of A_r in a classic way, by introducing a linear number of symbolic variables $V_r := \{v_q\}_{q\in Q}$ and a triple of formulas I_r, T_r, F_r where:

- $I_r := v_{q_0}$,
- $T_r := \bigwedge_{q\in Q}(v_q \rightarrow (\bigvee_{(q,a,q')\in\rho}(a \wedge v'_{q'})))$,
- $F_r := \bigvee_{q\in F} v_q$.

The FTS S_ϕ is defined as $\langle V_\phi, \mathcal{A}, T_\phi, I_\phi, F_\phi\rangle$, where $V_\phi = V_L \cup V_R$, $V_L := \{v_{qL}\}_{v_q\in V_r}$, $V_R := \{v_{qR}\}_{v_q\in V_r}$, $I_\phi := \top$, and

- $T_\phi := P_I \rightarrow I_r[v_{qL}/v_q]_{q\in Q} \wedge$
  $T_r[v_{qL}/v_q]_{q\in Q}[v'_{qL}/v'_q]_{q\in Q\setminus F}[v'_{qL} \vee P_F/v'_q]_{q\in F} \wedge$
  $(\bigwedge_{q\in Q} \neg v_{qR}) \rightarrow (\bigwedge_{q\in Q}(v'_{qL} \rightarrow v'_{qR})) \wedge$
  $T_r[v_{qR}/v_q]_{q\in Q}[v'_{qR}/v'_q]_{q\in Q\setminus F}[v'_{qR} \vee P_F/v'_q]_{q\in F} \wedge$
  $\bigwedge_{q\in Q}(v_{qR} \rightarrow v_{qL})$,
- $F_\phi := \bigwedge_{v_q\in V_r} \neg v_{qR}$.

By Definition 13, we can build an NBA A_ϕ from S_ϕ, such that $\mathcal{L}(A_\phi) = \mathcal{L}(S_\phi)$, and by the following theorem we can conclude $\mathcal{L}(S_\phi) = \mathcal{L}(\phi)$.

Theorem 6: $\mathcal{L}(A_\phi) = \mathcal{L}(\phi)$.

There are three main approaches to handle multiple fairness (see also Remark 1): the first is to reduce the fairness to a single constraint by considering a linear number of copies of the automaton (see for example [27]); the second is to delegate the problem to the search by considering the reachability of every constraint (such as in [28]); the third is to keep track of every satisfied constraint by considering one symbolic variable for each constraint (in a sense, MH works in this way). Here, we are considering the product of NBAs delegating the problem of handling multiple fairness to the search. Nevertheless the construction keeps some variables (the "right part" of the encoding) to track the fairness constraint of each NBA. Thus, keeping a set of fairness constraints may be inefficient. For this reason, we may consider a single global condition FC, shared by all automata corresponding to Suffix Conjunction Subformulas; in this case, we must change their transition relation so that the variables that track the fulfillment of the fairness condition are refreshed only when FC becomes true. Formally,

- $T_\phi := P_I \rightarrow I_r[v_{qL}/v_q]_{q\in Q} \wedge$
  $T_r[v_{qL}/v_q]_{q\in Q}[v'_{qL}/v'_q]_{q\in Q\setminus F}[v'_{qL} \vee P_F/v'_q]_{q\in F} \wedge$
  $FC \rightarrow (\bigwedge_{q\in Q}(v'_{qL} \rightarrow v'_{qR})) \wedge$
  $T_r[v_{qR}/v_q]_{q\in Q}[v'_{qR}/v'_q]_{q\in Q\setminus F}[v'_{qR} \vee P_F/v'_q]_{q\in F} \wedge$
  $\bigwedge_{q\in Q}(v_{qR} \rightarrow v_{qL})$,
- $FC := \bigwedge_{\phi \text{ is a Suffix Conjunction Formula}} F_\phi$.

VI. Experimental evaluation

We evaluated our methods within the NuSMV model checker [13]. We compared the monolithic approach (called MONO), the modular approach without optimizations (called MOD), and the modular approach with the optimizations of Section V-B.2 (called MODopt).

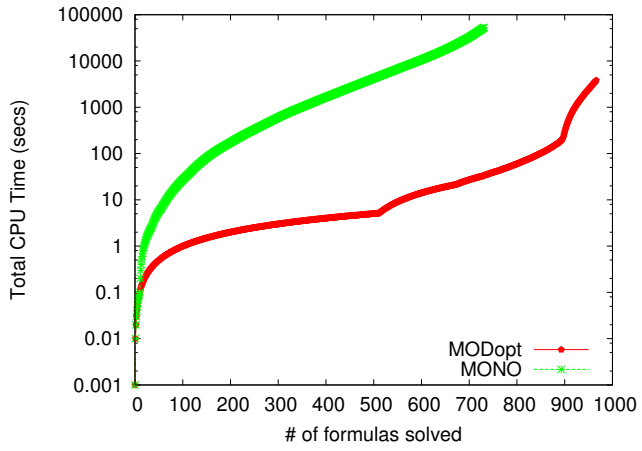
We compared the approaches with respect to automaton generation, fair cycle detection (i.e. language emptiness), and model checking, using both a BDD-based approach [28], and SBMC [29]. The BDD-based algorithm for fair cycle detection is restricted to the set of reachable states, which are preliminarily computed. The BDD algorithms were run with dynamic variable ordering activated. For SBMC, we used MiniSAT [30] as SAT engine. These settings provided good performance for all the approaches.

For the comparison, we first generated a set of properties, applying to randomly generated SEREs typical patterns extracted from industrial case studies [14]. Then, we used both Boolean combinations and single and double implications between big conjunctions of typical properties. The latter cases model problems arising in requirements engineering setting, i.e. refinement and equivalence among specifications. Model checking tests have been performed using a number of PSL properties applied to the Gigamax model taken from the standard NuSMV distribution.

All experiments were run on a 3GHz Intel CPU equipped with 4GB of memory, with a time out of 900 seconds and a memory out of 1GB. The results are collected in Figure 1. We plot the number of problems solved in a given amount of time (the samples are ordered by increasing computation time). We consider the NBA *encoding time* starting from the PSL specifications, the *search time* to perform verification, and the *overall verification time* (the sum of the others two).

A. NBA encoding time.

We first compared the performance of the MONO and MODopt approaches on the encoding of the NBA, over a collection of 1000 properties. The results are summarized in the following plot.



The optimized modular approach MODopt clearly outperforms MONO. While MODopt is able to complete the generation for all the properties, MONO times out in more than 200 cases. In the cases where MONO terminated, MODopt is vastly more efficient: the monolithic approach is able to build the NBA for 700 properties in about 10^6 seconds, the modular approach dealt with almost all 1000 properties in less than 10^4 seconds, and with about 900 in less than 10^2 seconds.

We also compared the MOD approach, to evaluate the impact of the optimizations described in Section V-B.2. It turns out that such optimization are crucial: MOD is always inferior to MODopt. The main bottleneck in MOD appears to be the determinization step on the premises of suffix operator subformulas resulting from the normalization. For this reason, hereafter we consider only MONO and MODopt. Additional details are reported in [16].

B. Overall verification time.

We restricted the further experiments to a sample of 400 properties for which MONO succeeded on building the NBA in the given time limits. Figure 1(a) shows the results obtained performing language emptiness via BDDs. Both methods are not being able to complete the verification for all the properties within the given time limits. The plot shows that the times obtained with MODopt are worse than those with MONO. We conjecture that the bottleneck is twofold. First, MODopt tends to generate a higher number of fairness conditions. Second, differently from MONO, no semantic optimizations are carried out by MODopt on the LTL component. The gap is reduced when we consider total times (NBA encoding plus verification time), as shown in Figure 1(c). We remark that the worse performance are to be put in the right perspective: when MONO is unable to build the NBA, MODopt is actually the only chance to attempt the verification.

When using SBMC for language emptiness (on the same set of properties), the gap between MONO and MODopt is much smaller (Figure 1(b)), and is basically reverted when the total of encoding and search is considered (Figure 1(d)). We also notice that the MONO encoding is unable to complete the verification for about 20 (unsatisfiable) samples. Although more investigation is required, we conjecture that the SBMC algorithm may be able to exploit the higher number of fairness conditions, and the increased structure of the NBA generated by MODopt.

Figures 1(e) and 1(f) show, for model checking problems, the total times for BDD-based search and for SBMC. In the first case, there is no substantial difference between MONO and MODopt; in the second case, the advantages of MODopt seem to be confirmed.

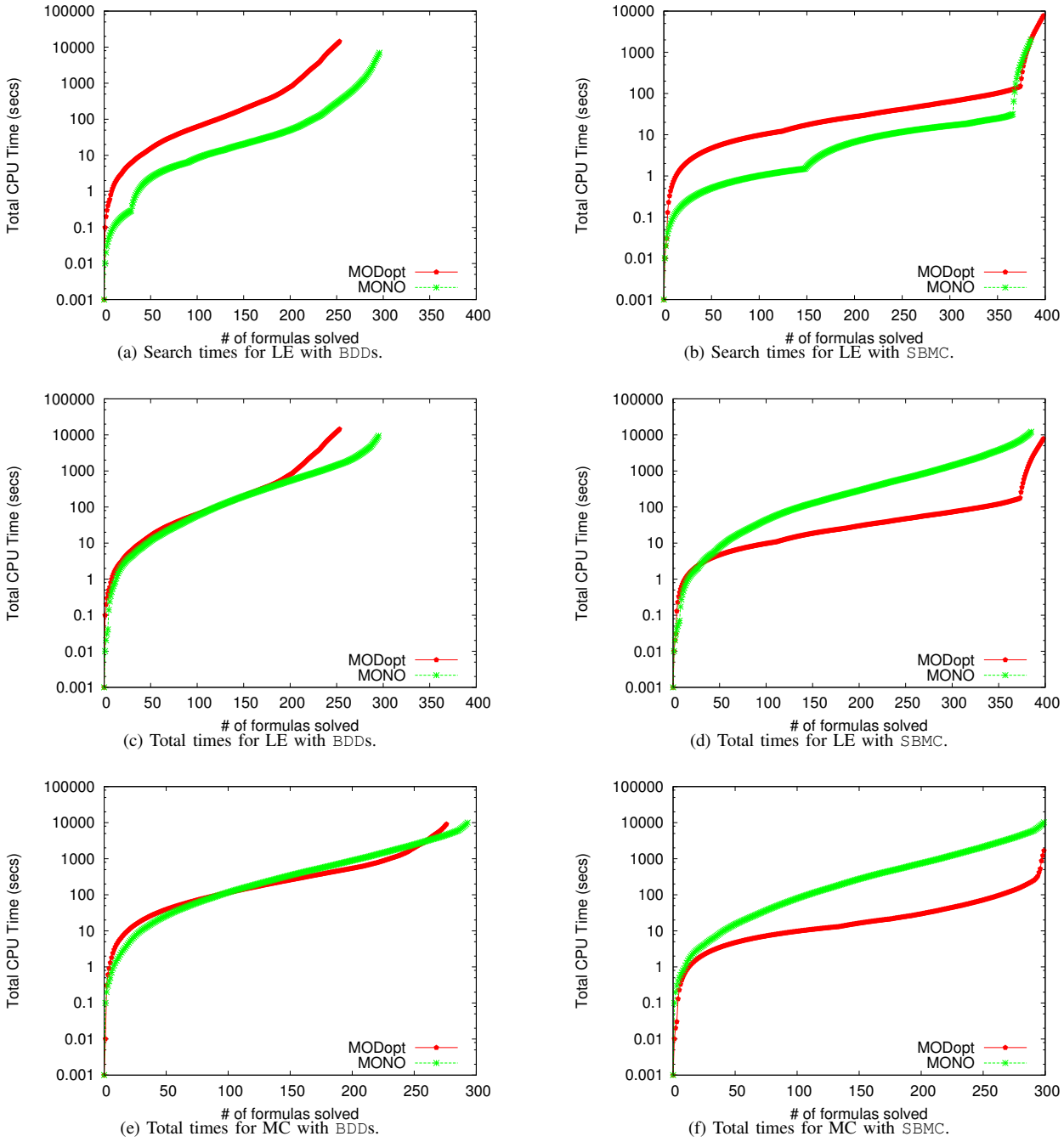
C. Discussion of the results.

The evaluation shows that the MODopt gives great gains in NBA encoding time.

When combined with SBMC, MODopt gives better overall performance than MONO, and in a few cases it is able to find a result when MONO times out.

In the case of BDD-based algorithms, if MONO is able to complete the generation of the NBA, then the results are better than MODopt.

We believe that there is space for further investigations and enhancements. In particular, the monolithic approach applies semantic optimizations on the explicit NBA before obtaining its symbolic encoding, that are made possible by the explicit combination of the automata for subformulas. The NBAs for the *Suffix Operator Subformulas* in the normalized formula are built in isolation; it seems possible to apply similar optimizations on top of the normalized structure, e.g. by aggressively simplifying the LTL part, detecting and exploiting possible relationships between the NBAs. Moreover, preliminary tests with BDDs have shown that the dynamic reordering takes a relevant portion of the search time, and that providing a good initial ordering can dramatically decrease the total time.



(a) Search times for LE with BDDs.
(b) Search times for LE with SBMC.
(c) Total times for LE with BDDs.
(d) Total times for LE with SBMC.
(e) Total times for MC with BDDs.
(f) Total times for MC with SBMC.

Fig. 1. Experimental evaluation results.

We also considered for comparison two recently proposed approaches, i.e. [15] and [7]. It was not possible to experimentally evaluate [15] since an implementation is not yet available. Given the similarities with our approach, however, we don't expect significant differences in performance.

As far as the method proposed in [7], which is tailored to SAT-based bounded model checking, an experimental comparison is currently ongoing. Detailed results will be included in the final version of [16].

VII. Conclusion

In this paper we have presented a new algorithm for the conversion of PSL into a symbolically represented NBA. The approach is based on the decomposition of the PSL specification into a normal form that separates out the LTL part and the SERE part. The various components can be independently generated, and are implicitly conjuncted. Additional optimizations are possible by exploiting the specific structure of subformulas involving suffix operators. The approach is proved

to be correct. A thorough experimental evaluation shows that the construction is extremely efficient, consuming much less resources than required by the monolithic construction. This makes it possible to tackle problems that were previously out of reach. The main limitation of the approach is currently that the generated automata have a redundant structure, which may result in degraded performance. In the future, we will work to find ways to mitigate this problem along the following directions. First, the structure of the automata could be used to define a better (initial) BDD variable ordering. Second, we plan to investigate the application of the reduction of liveness to safety checking [31], that typically results in shorter counterexamples, and can be useful in the particular case of fair cycle detection. Finally, we will investigate the use of reduction techniques (e.g. bisimulation minimization), which may result in smaller equivalent automata, thus reducing the search.

ACKNOWLEDGEMENT

The first three authors are Supported by the European Commission under contract 507219 (PROSYD).

The authors would like to thank Roderick Bloem, Ingo Pill and Moshe Vardi for fruitful discussions.

REFERENCES

[1] IEEE-Commission, "IEEE standard for Property Specification Language (PSL)," 2005, iEEE Std 1850-2005.

[2] A. Pnueli, "The temporal logic of programs," in *Proc. of 18th IEEE Symp. on Foundation of Computer Science*, 1977, pp. 46–57.

[3] D. Bustan, A. Flaisher, O. Grumberg, O. Kupferman, and M. Vardi, "Regular Vacuity," in *CHARME*, 2005, pp. 191–206.

[4] P. Gastin and D. Oddoux, "Fast ltl to büchi automata translation," in *Computer Aided Verification, Proc. of 13th International Conference*, ser. LNCS, vol. 2102. Springer-Verlag, 2001, pp. 53–65.

[5] C. Fritz, "Constructing Büchi Automata from Linear Temporal Logic Using Simulation Relations for Alternating Büchi Automata," in *Proc. of the 8th International Conference on Implementation and Application of Automata*, ser. LNCS, vol. 2759, 2003, pp. 35 – 48.

[6] S. Ben-David, R. Bloem, D. Fisman, A. Griesmayer, I. Pill, and S. Ruah, "Automata Construction Algorithms Optimized for PSL," http://www.prosyd.org, 2005, PROSYD deliverable D 3.2/4.

[7] K. Heljanko, T. Junttila, M. Keinänen, M. Lange, and T. Latvala, "Bounded Model Checking for Weak Alternating Büchi Automata," in *Proc. of the 18th Int. Conf. on Computer Aided Verification, CAV'06*, ser. LNCS, vol. 4144, Seattle (USA), 2006, pp. 95–108.

[8] S. Miyano and T. Hayashi, "Alternating finite automata on omega-words," *Theor. Comput. Sci.*, vol. 32, pp. 321–330, 1984.

[9] R. Bloem, A. Cimatti, I. P. ad M. Roveri, and S. Semprini, "Symbolic Implementation of Alternating Automata," in *Proc. of* 11^{th} *International Conference on Implementation and Application of Automata (CIAA06).*, ser. LNCS, vol. 4094, 2006, pp. 208–218.

[10] D. E. Muller, A. Saoudi, and P. E. Schupp, "Alternating automata, the weak monadic theory of trees and its complexity." *Theor. Comput. Sci.*, vol. 97, no. 2, pp. 233–244, 1992.

[11] F. Somenzi and R. Bloem, "Efficient Büchi Automata from LTL Formulae," in *Proc. of the 12th International Conference on Computer-Aided Verification*, ser. LNCS, vol. 1855. Springer-Verlag, 2000, pp. 247–263.

[12] R. Sebastiani, S. Tonetta, and M. Vardi, "Symbolic Systems, Explicit Properties: On Hybrid Approaches for LTL Symbolic Model Checking," in *Proc. of the 16th International Conference on Computer-Aided Verification (CAV'05)*, 2005, pp. 350–363.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NUSMV: a new Symbolic Model Verifier," in *Proc. of the 11th International Conference on Computer-Aided Verification*, ser. LNCS, vol. 1633. Springer-Verlag, 1999, pp. 495 – 499.

[14] S. B. David and A. Orni, "Property-by-Example guide: a handbook of PSL/Sugar examples," http://www.prosyd.org, 2005, PROSYD deliverable D 1.1/3.

[15] A. Pnueli and A. Zaks, "PSL Model Checking and Run-time Verification via Testers," in *Proc. of* 14^{th} *International Symposium on Formal Methods (FM'06)*, ser. LNCS, vol. 4085, Hamilton, Ontario, Canada, 2006, pp. 573–586.

[16] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta, "From PSL to NBA: a Modular Symbolic Encoding," ITC-irst, Tech. Rep. 18-08-06, August 2006.

[17] R. Armoni, D. Bustan, O. Kupferman, and M. Y. Vardi, "Resets vs. aborts in linear temporal logic." in *TACAS*, 2003, pp. 65–80.

[18] O. Lichtenstein, A. Pnueli, and L. Zuck, "The Glory of the Past," in *Logic of Programs*, 1985, pp. 196–218.

[19] O. Kupferman and M. Y. Vardi, "Weak alternating automata are not that weak." in *ISTCS*, 1997, pp. 147–158.

[20] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems, Specif ication*. New York: Springer Verlag, 1992.

[21] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[22] H. Yamamoto, "An Automata-Based Recognition Algorithm for Semi-extended Regular Expressions," in *MFCS*, 2000, pp. 699–708.

[23] O. Kupferman and S. Zuhovitzky, "An Improved Algorithm for the Membership Problem for Extended Regular Expressions," in *MFCS*, 2002, pp. 446–458.

[24] M. Vardi, "An Automata-Theoretic Approach to Linear Temporal Logic," in *Banff Higher Order Workshop*, 1995, pp. 238–266.

[25] E. Clarke, O. Grumberg, and K. Hamaguchi, "Another Look at LTL Model Checking," *Formal Methods in System Design*, vol. 10, no. 1, pp. 47–71, 1997.

[26] E. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 1999.

[27] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory-Efficient Algorithms for the Verification of Temporal Properties," *Formal Methods in System Design*, vol. 1, no. 2/3, pp. 275–288, 1992.

[28] E. Emerson and C. Lei, "Efficient Model Checking in Fragments of the Propositional μ-Calculus," in *Proc. of the Symposium on Logic in Computer Science*. IEEE Computer Society, 1986, pp. 267–278.

[29] K. Heljanko, T. A. Junttila, and T. Latvala, "Incremental and complete bounded model checking for full PLTL," in *Proc. of the 17th Int. Conf. on Computer Aided Verification (CAV'05)*, ser. LNCS, vol. 3576. Springer, 2005, pp. 98–111.

[30] N. Eén and N. Sörensson, "MiniSAT," 2005, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html.

[31] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 5, no. 2–3, pp. 185–204, 2004.

Assume-Guarantee Reasoning for Deadlock

Sagar Chaki, Software Engineering Institute
Nishant Sinha, Carnegie Mellon University
chaki@sei.cmu.edu nishants@cs.cmu.edu

***Abstract*— We extend the learning-based automated assume guarantee paradigm to perform compositional deadlock detection. We define Failure Automata, a generalization of finite automata that accept regular failure sets. We develop a learning algorithm L^F that constructs the minimal deterministic failure automaton accepting any unknown regular failure set using a minimally adequate teacher. We show how L^F can be used for compositional regular failure language containment, and deadlock detection, using non-circular and circular assume guarantee rules. We present an implementation of our techniques and encouraging experimental results on several non-trivial benchmarks.**

I. Introduction

Ensuring deadlock freedom is one of the most critical requirements in the design and validation of systems. The biggest challenge toward the development of effective deadlock detection schemes remains the *statespace explosion* problem. Compositional reasoning [1], [2], [3] is recognized to be one of the most promising approaches for alleviating statespace explosion. This paper presents an automated compositional deadlock detection procedure based on assume-guarantee (AG) [4] reasoning.

In general, AG reasoning revolves around a proof rule that relates system components and assumptions about them to global system properties. In order to apply the proof rule, one is normally required to construct manually appropriate assumptions that can discharge the premises of the rule. In most realistic situations however, suitable assumptions are quite complicated and the absence of automated assumption generation techniques has been a major stumbling block toward the wider practical adoption of AG reasoning.

An important breakthrough in this respect has been the use of learning algorithms for assumption construction [5]. The general idea is to learn an automaton corresponding to the *weakest assumption* [6] that can discharge the AG premises. The learning process is embedded in the overall verification procedure in a way that guarantees termination with the correct result. The choice of the learning algorithm is dictated by the kind of automaton that can represent the weakest assumption, which in turn depends on the verification goal. For example, in the case of trace containment [5], weakest assumptions are naturally represented as deterministic *finite* automata, and this leads to the use of the L^* [7] learning algorithm. Similarly, in the case of simulation [8], the corresponding choices are deterministic *tree* automata and the L^T learning algorithm.

However, neither of the above two options are appropriate for deadlock detection. Intuitively, word (as well as tree) automata are unable to capture *failures* [9], a critical concept for understanding, and detecting, deadlocks. Note that it is possible to devise schemes for transforming any deadlock detection problem to one of ordinary trace containment. However, such schemes invariably introduce new components and an exponential number of actions, and are thus not scalable. Our work, therefore, was initiated by the search for an appropriate automata-theoretic formalism that can handle failures directly. Our overall contribution is a deadlock detection algorithm that uses learning-based automated AG reasoning, and does not require the introduction of additional actions or components.

As we shall see, two key ingredients of our solution are: (i) a new type of acceptors for regular failure languages with a non-standard accepting condition, and (ii) a notion of parallel composition between these acceptors that is consistent with the parallel composition of the languages accepted by them. The accepting condition we use is novel, and employs a notion of maximality to crucially avoid the introduction of an exponential number of new actions. To the best of our knowledge, such acceptors and their composition have not been discussed before. In addition, we believe that this paper presents the first use of learning in the context of automated AG reasoning for deadlock detection. More specifically, we make the following contributions.

First, we present the theory of *regular* failure languages (RFLs) which are *downward-closed*, and define failure automata that exactly accept the set of regular failure languages. Although RFLs are closed under union and intersection, they are not closed under complementation, an acceptable price we pay for using the notion of maximality. Further, we show a Myhill-Nerode-like theorem for RFLs and failure automata. **Second**, we show that the failure language of an LTS M is regular and checking deadlock-freedom for M is a particular instance of the problem of checking containment of RFLs. We present an algorithm for checking containment of RFLs. Note that checking containment of a failure language L_1 by a failure language L_2 is not possible in the usual way by complementing L_2 and intersecting with L_1 since RFLs are not closed under complementation. **Third**, we present a sound and complete non-circular AG rule, called **AG-NC**, on failure languages for checking failure language specifications. Given failure languages L_1 and L_S, we define the weakest assumption failure language L_W: for every L_A, if $L_1 \parallel L_A \subseteq L_S$, then $L_A \subseteq L_W$. We then show, constructively, that if failure languages L_1 and L_2 are regular, then L_W uniquely exists, is also regular, and hence is accepted by a minimum failure automaton A_W. **Fourth**, we develop an algorithm L^F (pronounced "el-ef") to learn the minimum deterministic failure automaton that accepts an unknown regular failure language U using a minimally adequate teacher t

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

answer membership and candidate queries pertaining to U. We show how the teacher can be implemented using the RFL containment algorithm mentioned above. **Fifth**, we develop an automated and compositional deadlock detection algorithm that employs **AG-NC** and L^F. We also define a circular AG proof rule **AG-Circ** for deadlock detection and show how it can be used for automated and compositional deadlock detection. **Finally**, we have implemented our approach in the COMFORT [10] reasoning framework. We present encouraging results on several non-trivial benchmarks, including an embedded OS, and Linux device drivers.

II. Related Work

Machine learning techniques have been used in several contexts related to verification [11], [12], [13], [14], [15]. We follow the approach of Cobleigh et al. [5] (respectively Chaki et al. [8]) to automate assume-guarantee reasoning for trace-containment (respectively simulation) between finite state systems (Alur et al. [16] have also investigated symbolic learning in this context). However, we apply this general paradigm for deadlock detection. Further, the L^F algorithm that we present may be of independent interest. The use of circular AG rules was also investigated in the context of trace containment by Barringer et al. [17].

Overkamp has explored synthesis of supervisory controller for discrete-event systems [18] based on failure semantics [9]. His notion of the least restrictive supervisor that guarantees deadlock-free behavior is similar to the weakest failure assumption in our case. However, our approach differs from his as follows: (i) we use failure automata to represent failure traces, (ii) we use learning to compute the weakest failure assumption automatically, and (iii) our focus is on checking deadlocks in software modules. Williams et al. [19] investigate an approach based on static analysis for detecting deadlocks that can be caused by incorrect lock manipulation by Java libraries, and also provide an excellent survey of related research. The problem of detecting deadlocks for pushdown programs communicating only via nested locking has been investigated by Kahlon et al. [20]. In contrast, we present a model checking based framework to compositionally verify deadlock freedom for non-recursive programs with arbitrary lock-based or rendezvous communication. Other non-compositional techniques for detecting deadlock have been investigated in context of partial-order reduction [21] and for checking refinement of CCS processes, using a more discriminative (than failure trace refinement) notion called stuck-free conformance [22].

Brookes and Roscoe [23] use the failure model to show the absence of deadlock in undirectional networks. They also generalize the approach to the class of conflict-free networks via decomposition and local deadlock analysis. In contrast, we provide a completely automated framework for detecting deadlocks in arbitrary networks of asynchronous systems using rendezvous communication. Our formalism is based on an automata-theoretic representation of failure traces. Moreover, in order to analyze the deadlock-freedom of a set of concurrent programs compositionally, we use both circular and non-circular assume-guarantee [4], [1], [17] rules. Amla et al. [24] have presented a sound and complete assume-guarantee method in the context of an abstract process composition framework. However, they do not discuss deadlock detection, nor explore the use of learning.

In the rest of this paper we omit proofs for the sake of brevity. Detailed proofs can be found in an extended version of this paper [25].

III. Failure Languages and Automata

In this section we present the theory of failure languages and failure automata. We consider a subclass of *regular* failure languages and provide a lemma relating regular failure languages and failure automata, analogous to Myhill-Nerode theorem for ordinary regular languages. We begin with a few standard [26] definitions.

Definition 1 (Labeled Transition System): A labeled transition system (LTS) is a quadruple $(S, Init, \Sigma, \delta)$ where: (i) S is a set of states, (ii) $Init \subseteq S$ is a set of initial states, (iii) Σ is a set of actions (alphabet), and (iv) $\delta \subseteq S \times \Sigma \times S$ is a transition relation.

We only consider LTSs such that both S and Σ are finite. We write $s \xrightarrow{\alpha} s'$ to mean $(s, \alpha, s') \in \delta$. A trace is any finite (possibly empty) sequence of actions, i.e., the set of all traces is Σ^*. We denote an empty trace by ϵ, a singleton trace $\langle\alpha\rangle$ by α, and the concatenation of two traces t_1 and t_2 by $t_1 \bullet t_2$. We extend the relation δ to a function $\widehat{\delta}$ on a set of states in the usual way. We also employ the usual definitions of determinism and completeness for LTSs.

Definition 2 (Finite Automaton): A finite automaton is a pair (M, F) such that $M = (S, Init, \Sigma, \delta)$ is an LTS and $F \subseteq S$ is a set of final states.

Let $G = (M, F)$ be a finite automaton. Then G is said to be deterministic (complete) iff the underlying LTS M is deterministic (complete).

Definition 3 (Refusal): Let $M = (S, Init, \Sigma, \delta)$ be an LTS and $s \in S$ be any state of M. We say that s refuses an action α iff $\forall s' \in S \,.\, (s, \alpha, s') \notin \delta$. We say that s refuses a set of actions R, and denote this by $Ref(s, R)$, iff s refuses every element of R. Note that the following holds: (i) $\forall s \,.\, Ref(s, \emptyset)$, and (ii) $\forall s, R, R' \,.\, Ref(s, R) \wedge R' \subseteq R \implies Ref(s, R')$, i.e., refusals are *downward-closed*.

Definition 4 (Failure): Let $M = (S, Init, \Sigma, \delta)$ be an LTS. A pair $(t, R) \in \Sigma^* \times 2^\Sigma$ is said to be a failure of M iff there exists some $s \in \widehat{\delta}(Init, t)$ such that $Ref(s, R)$. The set of all failures of an LTS M is denoted by $\mathcal{F}(M)$.

Note that a failure consists of both, a trace, and a refusal set. A (possibly infinite) set of failures L is said to be a failure language. Let us denote 2^Σ by $\widehat{\Sigma}$. Note that $L \subseteq \Sigma^* \times \widehat{\Sigma}$. Union and intersection of failure languages is defined in the usual way. The complement of L, denoted by $\overline{L}$, is defined to be $(\Sigma^* \times \widehat{\Sigma}) \setminus L$. A failure language is said to be *downward-closed* iff $\forall t \in \Sigma^* \,.\, \forall R \in \widehat{\Sigma} \,.\, (t, R) \in L \implies \forall R' \subseteq R.\ (t, R') \in L$. Note that in general, failure languages may not be downward closed. However, as we show later, failure languages generated from LTSs are always downward

because the refusal sets at each state of an LTS are downward-closed. In this article, we focus on downward-closed failure languages, in particular, *regular* failure languages.

Definition 5 (Deadlock): An LTS M is said to deadlock iff the following holds: $\mathcal{F}(M) \cap (\Sigma^* \times \{\Sigma\}) \neq \emptyset$. In other words, M deadlocks iff it has a reachable state that refuses every action in its alphabet.

Let us denote the failure language $\Sigma^* \times \{\Sigma\}$ by L_{Dlk}. Then, it follows that M is deadlock-free iff $\mathcal{F}(M) \subseteq \overline{L_{Dlk}}$.

Maximality. Let P be any subset of $\widehat{\Sigma}$. Then the set of maximal elements of P is denoted by $Max(P)$ and defined as follows: $Max(P) = \{R \in P \mid \forall R' \in P \centerdot R \not\subset R'\}$

For example, if $P = \{\{a\}, \{b\}, \{a,b\}, \{a,c\}\}$, then $Max(P) = \{\{a,b\}, \{a,c\}\}$. A subset P of $\widehat{\Sigma}$ is said to be *maximal* iff it is non-empty and $Max(P) = P$. Intuitively, failure automata are finite automata whose final states are labeled with *maximal* refusal sets. Thus, a failure (t, R) is accepted by a failure automaton M iff upon receiving input t, M reaches a final state labeled with a refusal R' such that $R \subseteq R'$. Note that the notion of maximality allows us to concisely represent downward-closed failure languages by using only the upper bounds of a set (according to subset partial order) to represent the complete set.

Definition 6 (Failure Automaton): A failure automaton (FLA) is a triple (M, F, μ) such that $M = (S, Init, \Sigma, \delta)$ is an LTS, $F \subseteq S$ is a set of final states, and $\mu : F \rightarrow 2^{\widehat{\Sigma}}$ is a mapping from the final states to $2^{\widehat{\Sigma}}$ such that: $\forall s \in F \centerdot \mu(s) \neq \emptyset \wedge \mu(s) = Max(\mu(s))$.

Let $A = (M, F, \mu)$ be a FLA. Then A is said to be deterministic (respectively complete) iff the underlying LTS M is deterministic (respectively complete).

Definition 7 (Language of a FLA): Let $A = (M, F, \mu)$ be a FLA such that $M = (S, Init, \Sigma, \delta)$. Then a failure (t, R) is accepted by A iff $\exists s \in F \centerdot \exists R' \in \mu(s) \centerdot s \in \widehat{\delta}(Init, t) \wedge R \subseteq R'$. The language of A, denoted by $\mathcal{L}(A)$, is the set of all failures accepted by A.

Every deterministic FLA A can be extended to a complete deterministic FLA A' such that $\mathcal{L}(A') = \mathcal{L}(A)$ by adding a non-final *sink* state. In the rest of this article we consider FLA and languages over a fixed alphabet Σ. [1]

Lemma 1: A language is accepted by a FLA iff it is accepted by a deterministic FLA, i.e., deterministic FLA have the same accepting power as FLA in general.

Proof: (*Sketch*) By subset construction and properties of downward-closed sets. ∎

Regular Failure Languages (RFLs). A failure language is said to be *regular* iff it is accepted by some FLA. It follows from the definition of FLAs that RFLs are downward closed. Hence the set of RFLs is closed under union and intersection but not under complementation[2]. In addition, every regular failure language is accepted by an unique minimal deterministic FLA. The following Lemma is analogous to the Myhill-Nerode theorem for regular languages and ordinary finite automata.

Lemma 2: Every regular failure language(RFL) is accepted by a unique (up to isomorphism) minimal deterministic finite failure automaton.

Note that for any LTS M, $\mathcal{F}(M)$ is regular[3]. Indeed, the failure automaton corresponding to $M = (S, Init, \Sigma, \delta)$ is $A = (M, S, \mu)$ such that $\forall s \in S \centerdot \mu(s) = Max(\{R \mid Ref(s, R)\})$.

IV. Assume-Guarantee Reasoning for Deadlock

We now present an assume-guarantee style [4] proof rule for deadlock detection for systems composed of two components. We use the notion of parallel composition proposed in the theory of CSP [9] and define it formally.

Definition 8 (LTS Parallel Composition): Consider LTSs $M_1 = (S_1, Init_1, \Sigma_1, \delta_1)$ and $M_2 = (S_2, Init_2, \Sigma_2, \delta_2)$. Then the parallel composition of M_1 and M_2, denoted by $M_1 \amalg M_2$, is the LTS $(S_1 \times S_2, Init_1 \times Init_2, \Sigma_1 \cup \Sigma_2, \delta)$, such that $((s_1, s_2), \alpha, (s'_1, s'_2)) \in \delta$ iff the following holds: $\forall i \in \{1, 2\} \centerdot (\alpha \in \Sigma_i \wedge (s_i, \alpha, s'_i) \in \delta_i) \vee (\alpha \notin \Sigma_i \wedge s_i = s'_i)$.

Without loss of generality, we assume that both M_1 and M_2 have the same alphabet Σ. Indeed, any system with two components having different alphabets, say Σ_1 and Σ_2, can be converted to a bisimilar (and hence deadlock equivalent) system [8] with two components each having the same alphabet $\Sigma_1 \cup \Sigma_2$. Thus, all languages and automata we consider in the rest of this article will also be over the same alphabet Σ. We now extend the notion of parallel composition to failure languages. Observe that the composition involves set-intersection on the trace part and set-union on the refusal part of failures. Proofs of all the lemmas are in the full version [25] of the paper.

Definition 9 (Failure Language Composition): The parallel composition of any two failure languages L_1 and L_2, denoted by $L_1 \parallel L_2$, is defined as follows: $L_1 \parallel L_2 = \{(t, R_1 \cup R_2) \mid (t, R_1) \in L_1 \wedge (t, R_2) \in L_2\}$.

Lemma 3: For any failure languages L_1, L_2, L'_1 and L'_2, the following holds: $(L_1 \subseteq L'_1) \wedge (L_2 \subseteq L'_2) \implies (L_1 \parallel L_2) \subseteq (L'_1 \parallel L'_2)$.

Definition 10 (FLA Parallel Composition): Consider two FLAs $A_1 = (M_1, F_1, \mu_1)$ and $A_2 = (M_2, F_2, \mu_2)$. The parallel composition of A_1 and A_2, denoted by $A_1 \amalg A_2$[4], is defined as the FLA $(M_1 \amalg M_2, F_1 \times F_2, \mu)$ such that $\mu(s_1, s_2) = Max(\{R_1 \cup R_2 \mid R_1 \in \mu_1(s_1) \wedge R_2 \in \mu_2(s_2)\})$.

Note that we have used different notation ($\amalg$ and $\parallel$ respectively) to denote the parallel composition of automata and languages. Let M_1, M_2 be LTSs and A_1, A_2 be FLAs. Then the following two lemmas bridge the concepts of composition between automata and languages.

[1] FLA are closely related to automata on guarded strings [27], which contain arbitrary *transition* labels drawn from a partially-ordered set. In contrast, the *state* labels (refusals) in FLA are only maximal elements from such a set. Further, since it suffices to consider refusals at the end of a trace for checking deadlock freedom, we only label the final states of a FLA.

[2] For example, consider $\Sigma = \{\alpha\}$ and the RFL $L = \Sigma^* \times \{\emptyset\}$. Then $\overline{L} = \Sigma^* \times \{\{\alpha\}\}$ is not downward closed and hence is not an RFL.

[3] However, there exists RFLs that do not correspond to any LTS. In particular, any failure language L corresponding to some LTS must satisfy the following condition: $\exists R \subseteq \Sigma \centerdot (\epsilon, R) \in L$. Thus, the RFL $\{(\alpha, \emptyset)\}$ does not correspond to any LTS.

[4] We overload the operator $\amalg$ to denote parallel composition in the context of both LTSs and FLAs. The actual meaning of the operator will be clear from the context.

Lemma 4: $\mathcal{F}(M_1 \amalg M_2) = \mathcal{F}(M_1) \parallel \mathcal{F}(M_2)$.

Lemma 5: $\mathcal{L}(A_1 \amalg A_2) = \mathcal{L}(A_1) \parallel \mathcal{L}(A_2)$.

Regular Failure Language Containment (RFLC). We develop a general compositional framework for checking regular failure language containment. This framework is also applicable to deadlock detection since, as we illustrate later, deadlock freedom is a form of RFLC. Recall that regular failure languages are not closed under complementation and hence, given RFLs L_1 and L_2, it is not possible to verify $L_1 \subseteq L_2$ in the usual manner, by checking if $L_1 \cap \overline{L_2} = \emptyset$. However, as is shown by the following crucial lemma, it is possible to check containment between RFLs using their representations in terms of deterministic FLA, without having to complement the automaton corresponding to L_2.

Lemma 6: Consider any FLA A_1 and A_2. Let $A_1' = (M_1, F_1, \mu_1)$ and $A_2' = (M_2, F_2, \mu_2)$ be the FLA obtained by determinizing A_1 and A_2 respectively, and let $M_1 = (S_1, Init_1, \Sigma, \delta_1)$ and $M_2 = (S_2, Init_2, \Sigma, \delta_2)$. Then $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$ iff for every reachable state (s_1, s_2) of $M_1 \amalg M_2$ the following condition holds: $s_1 \in F_1 \implies (s_2 \in F_2 \wedge (\forall R_1 \in \mu_1(s_1) \,.\, \exists R_2 \in \mu_2(s_2) \,.\, R_1 \subseteq R_2))$.

In other words, we can check if $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$ by determinizing A_1 and A_2, constructing the *product* of the underlying LTSs and checking if the condition in Lemma 6 holds on every reachable state of the product. The condition essentially says that for every reachable state (s_1, s_2), if s_1 is final, then s_2 is also final and each refusal R_1 labeling s_1 is contained in some refusal R_2 labeling s_2.

Now suppose that $\mathcal{L}(A_1)$ is obtained by composing two RFLs L_1 and L_2, i.e., $\mathcal{L}(A_1) = L_1 \parallel L_2$ and let $\mathcal{L}(A_2) = L_S$, the specification language. In order to check RFLC between $L_1 \parallel L_2$ and L_S, the approach presented in lemma 6 will require us to directly compose L_1, L_2 and L_S, a potentially expensive computation. In the following, we first show that checking deadlock-freedom is a particular case of RFLC and then present a compositional technique to check RFLC (and hence deadlock-freedom) that avoids composing L_1 and L_2 (or their FLA representations) directly.

Deadlock as Regular Failure Language Containment. Given three RFLs L_1, L_2 and L_S, we can use our regular language containment algorithm to verify whether $(L_1 \parallel L_2) \subseteq L_S$. If this is the case, then our algorithm returns TRUE. Otherwise it returns FALSE along with a counterexample $CE \in (L_1 \parallel L_2) \setminus L_S$. Also, we assume that L_1, L_2 and L_S are represented as FLA. To use our algorithm for deadlock detection, recall that for any two LTSs M_1 and M_2, $M_1 \amalg M_2$ is deadlock free iff $\mathcal{F}(M_1 \amalg M_2) \subseteq \overline{L_{Dlk}}$. Let $L_1 = \mathcal{F}(M_1)$, $L_2 = \mathcal{F}(M_2)$ and $L_S = \overline{L_{Dlk}}$. Using Lemma 4, the above deadlock check reduces to verifying if $L_1 \parallel L_2 \subseteq L_S$. Observe that we can use our RFLC algorithm provided L_1, L_2 and L_S are regular. Recall that since M_1 and M_2 are LTSs, L_1 and L_2 are regular. Also, $\overline{L_{Dlk}}$ is regular since it is accepted by the failure automaton $A = (M, F, \mu)$ such that: (i) $M = (\{s\}, \{s\}, \Sigma, \delta)$, (ii) $\delta = \left\{ s \xrightarrow{\alpha} s \mid \alpha \in \Sigma \right\}$, (iii) $F = \{s\}$, and (iv) $\mu(s) = Max(\{R \mid R \subset \Sigma\})$. For instance, if $\Sigma = \{a, b, c\}$ then $\mu(s) = \{\{a, b\}, \{b, c\}, \{c, a\}\}$. Thus, deadlock detection is just a specific instance of RFLC.

Suppose we are given three RFLs L_1, L_2 and L_S in the form of their accepting FLA A_1, A_2 and A_S. To check $L_1 \parallel L_2 \subseteq L_S$, we can construct the FLA $A_1 \amalg A_2$ (cf. Lemma 10) and then check if $\mathcal{L}(A_1 \amalg A_2) \subseteq \mathcal{L}(A_S)$ (cf. Lemma 5 and 6). The problem with this naive approach is statespace explosion. In order to alleviate this problem, we present a compositional language containment scheme based on AG-style reasoning.

A Non-circular AG Rule. Consider RFLs L_1, L_2 and L_S. We are interested in checking whether $L_1 \parallel L_2 \subseteq L_S$. In this context, the following non-circular AG proof rule, which we call **AG-NC**, is both sound and complete:

$$\frac{L_1 \parallel L_A \subseteq L_S \qquad L_2 \subseteq L_A}{L_1 \parallel L_2 \subseteq L_S}$$

In principle, **AG-NC** enables us to prove $L_1 \parallel L_2 \subseteq L_S$ by discovering an assumption L_A that discharges its two premises. In practice, it leaves us with two critical problems. First, it provides no effective method for constructing an appropriate assumption L_A. Second, if no appropriate assumption exists, i.e., if the conclusion of **AG-NC** does not hold, then **AG-NC** does not help in obtaining a counterexample to $L_1 \parallel L_2 \subseteq L_S$. In this paper we develop and employ a learning algorithm that solves both the above problems. More specifically, our algorithm learns automatically, and incrementally, the *weakest* assumption L_W that can discharge the *first* premise of **AG-NC**. During this process, it is guaranteed to reach, in a finite number of steps, one of the following two situations, and thus always terminate with the correct result: **(1)** It discovers an assumption that can discharge *both* premises of **AG-NC**, and terminates with TRUE. **(2)** It discovers a counterexample CE to $L_1 \parallel L_2 \subseteq L_S$, and returns FALSE along with CE.

Weakest Assumption. Consider the proof rule **AG-NC**. For any L_1 and L_S, let $\widehat{L}$ be the set of all languages that can discharge the first premise of **AG-NC**. In other words, $\widehat{L} = \{L_A \mid (L_1 \parallel L_A) \subseteq L_S\}$. The following central theorem asserts that $\widehat{L}$ contains a unique weakest (maximal) element L_W that is also regular. This result is crucial for showing the termination of our approach.

Theorem 1: Let L_1 and L_S be any RFLs and f is a failure. Let us define a language L_W as follows: $L_W = \{f \mid (L_1 \parallel \{f\}) \subseteq L_S\}$. Then the following holds: (i) $L_1 \parallel L_W \subseteq L_S$, (ii) $\forall L \,.\, L_1 \parallel L \subseteq L_S \iff L \subseteq L_W$, and (iii) L_W is regular.

Proof: (*Sketch*) Parts (i) and (ii) can be proved from the definition of L_W. For (iii) we assume that L_1 and L_S are represented as failure automata A_1 and A_2, and use them to construct a failure automata A_W for L_W. The LTS for A_W is the *product* of the LTSs of A_1 and A_2. For every state (s_1, s_2), where s_1 and s_2 are final in their respective FLAs, we first compute a label X as follows: we add a refusal R to X iff for each refusal R_1 labeling s_1 there exists a refusal R_2 labeling s_2 such that $R_1 \cup R \subseteq R_2$. Finally, if $X \neq \emptyset$, we make (s_1, s_2) final and set $\mu(s_1, s_2) = Max(X)$. ■

Now that we have proved that the weakest environment assumption L_W is regular, we can apply a learning algorithm to iteratively construct a FLA assumption that accepts L_W. In particular, we develop a learning algorithm L^F that it

learns the minimal DFLA corresponding to L_W by asking queries about L_W to a minimally adequate teacher (MAT) and learning from them. In the next section, we present L^F. Subsequently, in Section VI, we describe how L^F is used in our compositional language containment procedure. A reader who is interested in the overall compositional deadlock detection algorithm more than the intricacies of L^F may skip directly to Section VI at this point.

V. Learning FLA

In this section we present an algorithm L^F to learn the minimal FLA that accepts an unknown RFL U. Our algorithm will use a minimally adequate teacher (MAT) that can answer two kinds of queries regarding U: **(1) Membership query:** Given a failure e the MAT returns TRUE if $e \in U$ and FALSE otherwise. **(2) Candidate query:** Given a deterministic FLA C, the MAT returns TRUE if $\mathcal{L}(C) = U$. Otherwise it returns FALSE along with a counterexample failure $CE \in (\mathcal{L}(C) \setminus U) \bigcup (U \setminus \mathcal{L}(C))$.

Observation Table. L^F uses an observation table to record the information it obtains by querying the MAT. The rows and columns of the table correspond to specific traces and failures respectively. Formally, a table is a triple $(\mathbb{T}, \mathbb{E}, \mathbb{R})$ where: (i) $\mathbb{T} \subseteq \Sigma^*$ is a set of traces, (ii) $\mathbb{E} \subseteq \Sigma^* \times \widehat{\Sigma}$ is a set of failures or experiments, and (iii) $\mathbb{R}$ is a function from $\widehat{\mathbb{T}} \times \mathbb{E}$ to $\{0, 1\}$ where $\widehat{\mathbb{T}} = \mathbb{T} \cup (\mathbb{T} \bullet \Sigma)$.

For any table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$, the function $\mathbb{R}$ is defined as follows: $\forall t \in \widehat{\mathbb{T}} \,.\, \forall e = (t', R) \in \mathbb{E}$, $\mathbb{R}(t, e) = 1$ iff $(t \bullet t', R) \in U$. Thus, given $\mathbb{T}$ and $\mathbb{E}$, algorithm L^F can compute $\mathbb{R}$ via membership queries to the MAT. For any $t \in \widehat{\mathbb{T}}$, we write $\mathbb{R}(t)$ to mean the function from $\mathbb{E}$ to $\{0, 1\}$ defined as follows: $\forall e \in \mathbb{E} \,.\, \mathbb{R}(t)(e) = \mathbb{R}(t, e)$.

An observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ is said to be well-formed iff: $\forall t_1 \in \mathbb{T} \,.\, \forall t_2 \in \mathbb{T} \,.\, t_1 \neq t_2 \implies \mathbb{R}(t_1) \neq \mathbb{R}(t_2)$. Essentially, this means that any two distinct rows t_1 and t_2 of a well-formed table can be distinguished by some experiment $e \in \mathbb{E}$. This also imposes an upper-bound on the number of rows of any well-formed table, as expressed by the following lemma.

Lemma 7: Let n be the number of states of the minimal DFLA accepting U and let $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ be any well-formed observation table. Then $|\mathbb{T}| \leq n$.

Closed observation table. An observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ is said to be *closed* iff it satisfies the following: $\forall t \in \mathbb{T} \,.\, \forall \alpha \in \Sigma \,.\, \exists t' \in \mathbb{T} \,.\, \mathbb{R}(t \bullet \alpha) = \mathbb{R}(t')$. Intuitively, this means that if we extend any trace $t \in \mathbb{T}$ by any action α then the result is indistinguishable from an existing trace $t' \in \mathbb{T}$ by the current set of experiments $\mathbb{E}$. Note that any well-formed table can be *extended* so that it is both well-formed and closed. This can be achieved by the algorithm **MakeClosed** shown in Figure 1. Observe that at every step of **MakeClosed**, the table $\mathcal{T}$ remains well-formed and hence, by Lemma 7, cannot grow infinitely. Also note that restricting the occurrence of refusals to $\mathbb{E}$ allows us to avoid considering the exponential possible refusal extensions of a trace while closing the table. Exponential number of membership queries will only be required if all possible refusals occur in $\mathbb{E}$.

Input: Well-formed observation table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$
while $\mathcal{T}$ is not closed **do**
  pick $t \in \mathbb{T}$ and $\alpha \in \Sigma$ such that $\forall t' \in \mathbb{T} \,.\, \mathbb{R}(t \bullet \alpha) \neq \mathbb{R}(t')$
  add $t \bullet \alpha$ to $\mathbb{T}$ and update $\mathbb{R}$ accordingly
return $\mathcal{T}$

Fig. 1. Algorithm **MakeClosed** extends an input well-formed table $\mathcal{T}$ so that the resulting table is both well-formed and closed.

Overall L^F algorithm. Algorithm L^F is iterative. It initially starts with a table $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ such that $\mathbb{T} = \{\epsilon\}$ and $\mathbb{E} = \emptyset$. Note that the initial table is well-formed. Subsequently, in each iteration L^F performs the following steps:

1) Make $\mathcal{T}$ closed by invoking **MakeClosed**.
2) Construct candidate DFLA C from $\mathcal{T}$ and make candidate query with C.
3) If the answer is TRUE, L^F terminates with C as the final answer.
4) Otherwise L^F uses the counterexample CE to the candidate query to add a single new failure to $\mathbb{E}$ and repeats from step 1.

In each iteration, L^F either terminates with the correct answer (step 3) or adds a new failure to $\mathbb{E}$ (step 4). In the latter scenario, the new failure to be added is constructed in a way that guarantees an upper bound on the total number of iterations of L^F. This, in turn, ensures its ultimate termination. We now present the procedures for: (i) constructing a candidate DFLA C from a closed and well-formed table $\mathcal{T}$ (used in step 2 above), and (ii) adding a new failure to $\mathbb{E}$ based on a counterexample to a candidate query (step 4).

Candidate construction. Let $\mathcal{T} = (\mathbb{T}, \mathbb{E}, \mathbb{R})$ be a closed and well-formed observation table. The candidate DFLA C is constructed from $\mathcal{T}$ as follows: $C = (M, F, \mu)$ and $M = (S, Init, \Sigma, \delta)$ such that: (i) $S = \mathbb{T}$, (ii) $Init = \{\epsilon\}$, (iii) $\delta = \left\{ t \xrightarrow{\alpha} t' \mid \mathbb{R}(t \bullet \alpha) = \mathbb{R}(t') \right\}$, (iv) $F = \{t \mid \exists e = (\epsilon, R) \in \mathbb{E} \,.\, \mathbb{R}(t, e) = 1\}$, and (v) $\mu(t) = Max(\{R \mid \mathbb{R}(t, (\epsilon, R)) = 1\})$.

Adding new failures. Let $C = (M, F, \mu)$ be a candidate DFLA such that $M = (S, Init, \Sigma, \delta)$. Let $CE = (t, R)$ be a counterexample to a candidate query made with C. In other words, $CE \in \mathcal{L}(C) \iff CE \notin U$. The algorithm **NewExp** adds a single new failure to $\mathcal{T}$ as follows. Let $t = \alpha_1 \bullet \ldots \bullet \alpha_k$. For $0 \leq i \leq k$, let t_i be the prefix of t of length i and t^i be the suffix of t of length $k - i$. In other words, for $0 \leq i \leq k$, we have $t_i \bullet t^i = t$.

Additionally, for $0 \leq i \leq k$, let s_i be the state of C reached by executing t_i. In other words, $s_i = \widehat{\delta}(t_i)$. Since the candidate C was constructed from an observation table $\mathcal{T}$, it corresponds to a row of $\mathcal{T}$, which in turn corresponds to a trace. Let us also refer to this trace as s_i. Finally, let $b_i = 1$ if the failure $\left(s_i \bullet t^i, R\right) \in U$ and 0 otherwise. Note that we can compute b_i by evaluating s_i and then making a membership query with $\left(s_i \bullet t^i, R\right)$. In particular, $s_0 = \epsilon$, and hence $b_0 = 1$ if $CE \in U$ and 0 otherwise. We now consider two cases.

Case 1: [$b_0 = 0$] In this case, there exists an in

$\{0, \ldots, k\}$ such that $b_j = 0$ and $b_{j+1} = 1$. L^F finds such an index j and adds the failure $\left(t^{j+1}, R\right)$ to $\mathbb{E}$. As a result, the table $\mathcal{T}$ becomes non-closed and therefore, the next candidate FLA has strictly more states than the current candidate C. Complete details can be found in the full version of this paper.

Case 2: [$b_0 = 1$] In this case, L^F adds a new failure to $\mathbb{E}$ that leads to the next candidate differing from the current candidate C in *at least one* of the following three ways: (i) it has strictly more states, (ii) it has a new final state, and (iii) the labeling of one of the current final states gets augmented. Complete details can be found in the full version of this paper.

Correctness of L^F. Algorithm L^F always returns the correct answer in step 3 since it always does so after a successful candidate query. To see that L^F always terminates, observe that in every iteration, the candidate C computed by L^F undergoes at least one of the following three changes:

- **(Ch1)** The number of states of C, and hence the number of rows of the observation table $\mathcal{T}$, increases.
- **(Ch2)** The states and transitions of C remain unchanged but a state of C that was previously non-final becomes final.
- **(Ch3)** The states, transitions and final states of C remain unchanged but for some final state s of C, the size of $\mu(s)$ increases.

Of the above changes, **Ch1** can happen at most n times where n is the number of states of the minimal DFLA accepting U. Between any two consecutive occurrences of **Ch1**, there can only be a finite number of occurrences of **Ch2** and **Ch3**. Hence there can only be a finite number of iterations of L^F. Therefore, L^F always terminates.

Number of iterations. To analyze the complexity of L^F we have to impose a tighter bound on the number of iterations. We already know that **Ch1** can happen at most n times. Since a final state can never become non-final, **Ch2** can also occur at most n times. Now let the minimal DFLA accepting U be $A = (M, F, \mu)$ such that $M = (S, Init, \Sigma, \delta)$. Consider the set $P = \bigcup_{s \in F} \mu(s)$ and let $n' = |P|$. Since each **Ch3** adds an element to $\mu(s)$ for some $s \in F$, the total number of occurrences of **Ch3** is at most n'. Therefore the maximum number of iterations of L^F is $2n + n' = \mathcal{O}(n + n')$.

Time complexity. Let us make the standard assumption that each MAT query takes $\mathcal{O}(1)$ time. From the above discussion we see that the number of columns of the observation table is at most $\mathcal{O}(n + n')$. The number of rows is at most $\mathcal{O}(n)$. Let us assume that the size of Σ is a constant. Then the number of membership queries, and hence time, needed to fill up the table is $\mathcal{O}(n(n + n'))$.

Let m be the length of the longest counterexample returned by a candidate query. Then to add each new failure, we have to make $\mathcal{O}(log(m))$ membership queries to find the appropriate index j. Also, let the time required to find the maximal element R_{max} be $\mathcal{O}(m')$. Then total time required for constructing each new failure is $\mathcal{O}((n + n')(log(m) + m'))$. Finally, the number of candidate queries equals the number of iterations and hence is $\mathcal{O}(n + n')$. Thus, in summary, we find that the time complexity of L^F is $\mathcal{O}((n + n')(n + log(m) + m'))$, which is polynomial in n, n', m and m'.

Space complexity. Let us again make the standard assumption that each MAT query takes $\mathcal{O}(1)$ space. Since the queries are made sequentially, total space requirement for all of them is still $\mathcal{O}(1)$. Also, the procedure for constructing a new failure can be performed in $\mathcal{O}(1)$ space. A trace corresponding to a table row can be $\mathcal{O}(n)$ long and there are $\mathcal{O}(n)$ of them. A failure corresponding to a table column can be $\mathcal{O}(m)$ long and there are $\mathcal{O}(n + n')$ of them. Space required to store the table elements is $\mathcal{O}(n(n + n'))$. Hence total space required for the observation table is $\mathcal{O}((n + m)(n + n'))$. Space required to store computed candidates is $\mathcal{O}(n^2)$. Therefore, the total space complexity is $\mathcal{O}((n + m)(n + n'))$ which is also polynomial in n, n' and m.

VI. Compositional Language Containment

Given RFLs L_1, L_2 and L_S (in the form of FLA that accept them) we want to check whether $L_1 \parallel L_2 \subseteq L_S$. If not, we also want to generate a counterexamples $CE \in (L_1 \parallel L_2) \setminus L_S$. To this end, we invoke the L^F algorithm to learn the weakest environment corresponding to L_1 and L_S. We present an implementation strategy for the MAT to answer the membership and candidate queries posed by L^F. In the following we assume that A_1, A_2 and A_S are the given FLAs such that $\mathcal{L}(A_1) = L_1$, $\mathcal{L}(A_2) = L_2$ and $\mathcal{L}(A_S) = L_S$.

Membership Query. The answer to a membership query with failure $e = (t, R)$ is TRUE if the following condition (which can be effectively decided) holds and FALSE otherwise: $\forall (t, R_1) \in L_1 \centerdot (t, R_1 \cup R) \in L_S$.

Candidate Query. A candidate query with a failure automaton C is answered step-wise as follows:

1) Check if $\mathcal{L}(A_1 \amalg C) \subseteq \mathcal{L}(A_S)$. If not, let $(t, R_1 \cup R)$ be the counterexample obtained. Note that $(t, R) \in \mathcal{L}(C) \setminus U$. We return FALSE to L^F along with the counterexample (t, R). If $\mathcal{L}(A_1 \amalg C) \subseteq \mathcal{L}(A_S)$, we proceed to step 2.
2) Check if $\mathcal{L}(A_2) \subseteq \mathcal{L}(C)$. If so, we have obtained an assumption, viz., $\mathcal{L}(C)$, that discharges both premises of **AG-NC**. In this case, the overall language containment algorithm terminates with TRUE. Otherwise let (t', R') be the counterexample obtained. We proceed to step 3.
3) We check if there exists $(t', R'_1) \in \mathcal{L}(A_1)$ such that $(t', R'_1 \cup R') \notin \mathcal{L}(A_S)$. If so, then $(t', R'_1 \cup R') \in \mathcal{L}(A_1 \amalg A_2) \setminus \mathcal{L}(A_S)$ and the overall language containment algorithm terminates with FALSE and the counterexample $(t', R'_1 \cup R')$. Otherwise $(t', R') \in U \setminus \mathcal{L}(C)$ and we return FALSE to L^F along with the counterexample (t', R').

Note that in the above we are never required to compose A_1 with A_2. In practice, the candidate C (that we compose with A_1 in step 1 of the candidate query) is much smaller than A_2. Thus we are able to alleviate the statespace explosion problem. Also, note that our procedure will ultimately terminate with the correct result from either step 2 or 3 of the candidate query. This follows from the correctness of L^F algorithm: in the worst case, the candidate query will be made with a FLA C such that $\mathcal{L}(C) = L_W$. In this scenario, termination is guaranteed to occur due to Theorem 1.

Exp	LOC	C	St	No Deadlock								Deadlock							
				Plain		AG-NC			AG-Circ			Plain		AG-NC			AG-Circ		
				T	M	T	M	A	T	M	A	T	M	T	M	A	T	M	A
MC	7272	2	2874	-	*	308	**903**	**5**	**307**	**903**	6	372	β	386	980	**13**	**313**	**979**	16
MC	7272	3	2874	-	*	766	**1155**	**11**	**459**	**1155**	12	-	-	-	-	-	-	-	-
MC	7272	4	2874	-	*	*	1453	-	**716**	**1453**	24	-	-	-	-	-	-	-	-
ide	18905	3	672	571	*	338	50	**11**	**62**	**47**	12	755	*	*	80	-	**557**	**551**	**125**
ide	18905	4	716	972	*	*	63	-	**195**	**55**	**24**	978	*	*	84	-	2913	*	-
ide	18905	5	760	1082	*	*	84	-	**639**	**85**	**48**	1082	*	*	89	-	*	498	-
syn	17262	4	117	733	*	1547	**19**	**21**	**58**	21	24	864	*	**127**	**181**	**2**	133	**181**	6
syn	17262	5	127	713	*	*	19	-	**224**	**47**	**48**	1088	*	844	*	-	867	*	-
syn	17262	6	137	767	*	*	27	-	**1815**	**189**	**96**	-	*	1188	*	-	-	*	-
mx	15717	3	1995	1154	*	2079	140	**11**	**639**	**123**	12	1182	*	657	**364**	**2**	**630**	**364**	5
mx	15717	4	2058	1545	*	-	168	-	**713**	**139**	24	1309	*	1627	*	-	1206	*	-
mx	15717	5	2121	1660	*	-	179	-	**2131**	**185**	48	-	*	3368	*	-	2276	*	-
tg3	36774	3	1653	971	*	1568	118	**11**	**406**	**111**	12	894	*	**486**	**393**	**2**	499	**393**	5
tg3	36774	4	1673	927	*	-	149	-	**486**	**131**	24	1096	*	1036	*	-	1037	*	-
tg3	36774	5	1693	1086	*	-	158	-	**1338**	**165**	48	-	*	2186	*	-	1668	*	-
tg3	36774	6	1713	1252	*	-	157	-	**3406**	**313**	96	1278	*	*	-	-	1954	*	-
IPC	818	3	302	**195**	α	703	**338**	**49**	478	355	**49**	-	-	-	-	-	-	-	-
DP	82	6	30	274	*	**100**	**330**	11	286	414	**9**	-	-	-	-	-	-	-	-
DP	109	8	30	302	*	**1551**	**565**	**11**	*	1474	-	-	-	-	-	-	-	-	-

TABLE I

EXPERIMENTAL RESULTS. C = # OF COMPONENTS; ST = # OF STATES OF LARGEST COMPONENT; T = TIME (SECONDS); M = MEMORY (MB); A = # OF STATES OF LARGEST ASSUMPTION; * = RESOURCE EXHAUSTION; - = DATA UNAVAILABLE; $\alpha = 1247$; $\beta = 1708$. BEST FIGURES ARE HIGHLIGHTED.

VII. ARBITRARY COMPONENTS AND CIRCULARITY

We investigated two approaches for handling more than two components. First, we applied **AG-NC** recursively. This can be demonstrated for languages L_1, L_2, L_3 and L_S by the following proof-rule.

$$\frac{L_1 \parallel L_A^1 \subseteq L_S \qquad \dfrac{L_2 \parallel L_A^2 \subseteq L_A^1 \qquad L_3 \subseteq L_A^2}{L_2 \parallel L_3 \subseteq L_A^1}}{L_1 \parallel L_2 \parallel L_3 \subseteq L_S}$$

At the top-level, we apply **AG-NC** on the two languages L_1 and $L_2 \parallel L_3$. Now the second premise becomes $L_2 \parallel L_3 \subseteq L_A^1$ and we can again apply **AG-NC**. In terms of the implementation of the MAT, the only difference is in step 2 of the candidate query (cf. Section VI). More specifically, we now invoke the language containment procedure recursively with $\mathcal{L}(A_2)$, $\mathcal{L}(A_3)$ and $\mathcal{L}(C)$ instead of checking directly for $\mathcal{L}(A_2) \subseteq \mathcal{L}(C)$. This technique can be extended to any finite number of components.

Circular AG Rule. We also explored a circular AG rule. Unlike **AG-NC** however, the circular rule is specific to deadlock detection and not applicable to language containment in general. For any RFL L let us write $W(L)$ to denote the weakest assumption against which L does not deadlock. In other words, $\forall L' \,.\, L \parallel L' \subseteq \overline{L_{Dlk}} \iff L' \subseteq W(L)$. It can be shown that: **(PROP)** $\forall t \in \Sigma^* \,.\, \forall R \in \widehat{\Sigma} \,.\, (t,R) \in L \iff (t, \Sigma \setminus R) \notin W(L)$. The following theorem provides a circular AG rule for deadlock detection.

Theorem 2: Consider any two RFLs L_1 and L_2. Then the following proof rule, which we call **AG-Circ**, is both sound and complete.

$$\frac{L_1 \parallel L_A^1 \subseteq \overline{L_{Dlk}} \qquad L_2 \parallel L_A^2 \subseteq \overline{L_{Dlk}} \qquad W(L_A^1) \parallel W(L_A^2) \subseteq \overline{L_{Dlk}}}{L_1 \parallel L_2 \subseteq \overline{L_{Dlk}}}$$

Implementation. To use this rule for deadlock detection for two components L_1 and L_2 we use the following iterative procedure:

1) Using the first premise, construct a candidate C_1 similar to Step 1 of the candidate query in **AG-NC** (cf. Section VI). Similarly, using the second premise, construct another candidate C_2. Construction of C_1 and C_2 proceeds exactly as in the case of **AG-NC**.
2) Check if $W(\mathcal{L}(C_1)) \parallel W(\mathcal{L}(C_2)) \subseteq \overline{L_{Dlk}}$. This is done either directly or via a compositional language containment using **AG-NC**. We compute the automata for $W(\mathcal{L}(C_1))$ and $W(\mathcal{L}(C_2))$ using the procedure described in the proof of Theorem 1. If the check succeeds then there is no deadlock in $L_1 \parallel L_2$ and we exit successfully. Otherwise, we proceed to Step 3.
3) From the counterexample obtained above construct $t \in \Sigma^*$ and $R \in \widehat{\Sigma}$ be such that $(t,R) \in W(\mathcal{L}(C_1))$ and $(t, \Sigma \setminus R) \in W(\mathcal{L}(C_2))$. Check if $(t,R) \in L_1$ and $(t, \Sigma \setminus R) \in L_2$. If both these checks pass then we have a counterexample t to the overall deadlock detection problem and therefore we terminate unsuccessfully. Otherwise, without loss of generality, suppose $(t,R) \notin L_1$. But then, from **PROP**, $(t, \Sigma \setminus R) \in W(L_1)$. Again from **PROP**, since $(t,R) \in W(\mathcal{L}(C_1))$, $(t, \Sigma \setminus R) \notin \mathcal{L}(C_1)$. This is equivalent to a failed candidate query for C_1 with counterexample $(t, \Sigma \setminus R)$, and we repeat from Step 1 above.

Note that even though we have presented **AG-Circ** in the context of only two components, it generalizes to an arbitrary, but finite, number of components.

VIII. EXPERIMENTAL VALIDATION AND CONCLUSION

We implemented our algorithms in the COMFORT [10] reasoning framework and experimented with a set of real-life examples. All our experiments were done on a 2

Pentium 4 machine running RedHat 9 and with time limit of 1 hour and a memory limit of 2 GB. Our results are summarized in Table I. The *MC* benchmarks are derived from Micro-C version 2.70, a lightweight OS for real-time embedded applications. The *IPC* benchmark is based on an inter-process communication library used by an industrial robot controller software. The *ide*, *syn*, *mx* and *tg3* examples are based on Linux device drivers. Finally, *DP* is a synthetic benchmark based on the well-known dining philosophers example.

For each example, we obtained a set of benchmarks by increasing the number of components. For each such benchmark, we tested a version without deadlock, and another with an artificially introduced deadlock. In all cases, deadlock was caused by incorrect synchronization between components – the only difference was in the synchronization mechanism. Specifically, the dining philosophers synchronized using "forks". In all other examples, synchronization was achieved via a shared "lock".

For each benchmark, a finite LTS model was constructed via a predicate abstraction [10] that transformed the synchronization behavior into appropriate actions. For example, in the case of the *ide* benchmark, calls to the `spin_lock` and `spin_unlock` functions were transformed into *lock* and *unlock* actions respectively. Note that this makes sense because, for instance, multiple threads executing the driver for a specific device will acquire and release a common lock specific to that device by invoking `spin_lock` and `spin_unlock` respectively.

For each abstraction, appropriate predicates were supplied externally so that the resulting models would be precise enough to display the presence or absence of deadlock. In addition, care was taken to ensure that the abstractions were sound with respect to deadlocks, i.e., the extra behavior introduced did not eliminate any deadlock in the concrete system. Each benchmark was verified using explicit brute-force statespace exploration (referred to in Table I as "Plain"), the non-circular AG rule (referred as **AG-NC**), and the circular AG rule (referred as **AG-Circ**). When using **AG-Circ**, Step 2 (i.e., checking if $W(\mathcal{L}(C_1)) \parallel W(\mathcal{L}(C_2)) \subseteq \overline{L_{Dlk}}$) was done via compositional language containment using **AG-NC**.

We observe that the AG-based methods outperform the naive approach for most of the benchmarks. More importantly, for each benchmark, the growth in memory consumption with increasing number of components is benign for both AG-based approaches. This indicates that AG reasoning is effective in combating statespace explosion even for deadlock detection. We also note that larger assumptions (and hence time and memory) are required for detecting deadlocks as opposed to detecting deadlock freedom. Among the AG-based approaches, **AG-Circ** is in general faster than **AG-NC** but (on a few occasions) consumes negligible extra memory. In several cases, **AG-NC** runs out of time while **AG-Circ** is able to terminate successfully. Overall, whenever **AG-NC** and **AG-Circ** differ significantly in any real-life example, **AG-Circ** is superior.

Conclusion. We have extended the learning-based automated assume guarantee paradigm to deadlock detection. We have defined a new kind of automata that are similar to finite automata but accept failures instead of traces. We have also developed an algorithm, L^F, that learns the minimal failure automata accepting an unknown regular failure language using a minimally adequate teacher. We have shown how L^F can be used for compositional deadlock detection using both circular and non-circular assume-guarantee rules. Finally, we have implemented our technique and have obtained encouraging experimental results on several non-trivial benchmarks.

REFERENCES

[1] W. P. de Roever, H. Langmaack, and A. Pnueli, Eds., *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Revised Lectures*, ser. LNCS, vol. 1536. Springer, 1998.

[2] K. McMillan, "A compositional rule for hardware design refinement," in *Proc. of CAV*, 1997.

[3] O. Grumberg and D. E. Long, "Model checking and modular verification," *TOPLAS*, vol. 16, no. 3, pp. 843–871, May 1994.

[4] A. Pnueli, "In transition from global to modular temporal reasoning about programs," *Logics and models of concurrent systems*, vol. 13, pp. 123–144, 1985.

[5] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu, "Learning assumptions for compositional verification," in *Proc. of TACAS*, 2003, pp. 331–346.

[6] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer, "Assumption generation for software component verification," in *Proc. of ASE*, 2002.

[7] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, no. 2, pp. 87–106, 1987.

[8] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati, "Automated assume-guarantee reasoning for simulation conformance," in *Proc. of CAV*, 2005, pp. 534–547.

[9] C. A. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[10] S. Chaki, J. Ivers, N. Sharygina, and K. Wallnau, "The ComFoRT reasoning framework," in *Proc. of CAV*, 2005, pp. 164–169.

[11] D. Peled, M. Vardi, and M. Yannakakis, "Black box checking," in *Proc. of FORTE/PSTV*, October 1999.

[12] A. Groce, D. Peled, and M. Yannakakis, "Adaptive model checking," in *Proc. of TACAS*, 2002, pp. 357–370.

[13] R. Alur, P. Cerný, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes." in *Proc. of POPL*, 2005, pp. 98–109.

[14] P. Habermehl and T. Vojnar, "Regular model checking using inference of regular languages," in *Proc. of INFINITY'04*, 2004.

[15] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. of ICSE*, 1999.

[16] R. Alur, P. Madhusudan, and W. Nam, "Symbolic compositional verification by learning assumptions," in *Proc. of CAV*, 2005.

[17] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu, "Proof rules for automated compositional verification," in *Proc. of the 2nd Workshop on SAVCBS*, 2003.

[18] A. Overkamp, "Supervisory control using failure semantics and partial specifications," *Automatic Control, IEEE Transactions on*, vol. 42, no. 4, pp. 498–510, April 1997.

[19] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *Proc. of ECOOP*, 2005, pp. 602–629.

[20] V. Kahlon, F. Ivancic, and A. Gupta, "Reasoning about threads communicating via locks." in *Proc. of CAV*, 2005, pp. 505–518.

[21] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[22] C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof, "Stuck-free conformance." in *CAV*, 2004, pp. 242–254.

[23] S. D. Brookes and A. W. Roscoe, "Deadlock analysis in networks of communicating processes." *Distributed Computing*, vol. 4, pp. 209–230, 1991.

[24] N. Amla, E. A. Emerson, K. S. Namjoshi, and R. J. Trefler, "Abstract patterns of compositional reasoning." in *Proc. of CONCUR*, 2003, pp. 423–438.

[25] S. Chaki and N. Sinha, "Assume-guarantee reasoning for deadlock," Software Engineering Institute, Pittsburgh, PA, Technical note CMU/SEI-2006-TN-028, 2006.

[26] A. W. Roscoe, *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[27] D. Kozen, "Automata on guarded strings and applications," in *Matematica Contempornea 24 (2003)*, pp. 117–139.

A Refinement Method for Validity Checking of Quantified First-Order Formulas in Hardware Verification

Husam Abu-Haimed
Nusym Technology, Inc
Los Gatos
California 95032
husam@stanfordalumni.org

David L. Dill
Computer Science Department
Stanford University
California 94305
dill@cs.stanford.edu

Sergey Berezin
Synopsys, Inc
Mountain View
California 94043
berezin@synopsys.com

***Abstract*— We introduce a heuristic for automatically checking the validity of first-order formulas of the form $\forall\alpha^m\ \exists\beta^n.\ \Psi(\alpha^m, \beta^n)$ that are encountered in inductive proofs of hardware correctness. The heuristic introduced in this paper is used to automatically check the validity of k-step induction formulas needed to verify hardware designs. The heuristic works on word-level designs that can have data and address buses of arbitrary widths. Our refinement heuristic relies on the idea of *predicate instantiation* introduced in [2]. The heuristic proves quantified formulas by the use of a validity checker, CVC [21], and a first-order theorem prover, Otter [16]. Our heuristic can be used as a stand-alone technique to verify word-level designs or as a component in an interactive theorem prover. We show the effectiveness of this heuristic for hardware verification by verifying a number of hardware designs completely automatically. The large size of the quantified formulas encountered in these examples shows the effectiveness of our heuristic as a component of a theorem prover.**

I. Introduction

The complexity of most industrial hardware designs are beyond the capacity of formal verification tools. Compositional techniques have been used to decompose large designs into smaller ones and model checking each separately [17], [18]. The task of decomposing hardware designs, however, requires significant effort and detailed knowledge and understanding of the design.

Abstraction has been used to transform complex and possibly infinite systems into small finite systems that can be model checked [11], [12], [8]. Abstraction techniques have been used successfully to verify many designs. The majority of real designs, however, are still beyond the capacity of such tools.

Theorem proving techniques have been used for a long time to verify complex systems containing complex functional units, wide busses, and large memories [19], [9], [13]. Those techniques, however, require significant effort and experience from the verification engineer to guide the tool. That makes the use of theorem proving very limited in verification in the hardware industry.

Validity checkers for decidable segments of first order logic have been used successfully in many hardware verification techniques. Such logics were used to verify pipelined microprocessors [7] and superscalar microprocessors [6], [24], [15] with a high degree of success. Verification of infinite systems within these logics is usually done by induction on time. That requires a lot of manual effort to find and strengthen invariants needed for the induction proof to go through. Many techniques have been proposed to automate the process of finding and strengthening invariants [22], [5], [4], [23]. The process, however, is still mostly manual.

In [2], the method of *symbolic consistency testing* was introduced to strengthen invariants by the use of k-step induction and then proving those formulas by a validity checker, CVC [21]. Induction formulas, however, contain existential quantifiers which cannot be handled by validity checkers for quantifier-free logics. In [2], the idea of *predicate instantiation* (see section II-B) was introduced to eliminate existential quantifiers. In this method, the user provides symbolic test vectors which are used as instantiation predicates to eliminate existential quantifiers. Finding those vectors can be challenging for complex systems.

In this paper we introduce an automatic refinement heuristic for checking the validity of such k-step induction formulas by automatically finding the instantiation predicates. This heuristic is not limited to k-step induction formulas, but can be used to check the validity of any formula of the form $\forall\alpha^m\ \exists\beta^n\ \Psi(\alpha^m, \beta^n)$. Hence, our heuristic can be used as a method for hardware verification or as a component of an interactive theorem prover. We show the effectiveness of this heuristic on a number of hardware examples that were considered in [2]. The examples are various architectures of cached memory systems and a simple pipelined processor. In [2], the instantiation predicates were found manually. The heuristic we introduce in this paper finds those predicates and verifies those designs completely automatically. The large sizes of the formulas encountered in these examples also shows the effectiveness of the heuristic in other applications of theorem proving. The formulas were generated by word-level simulation (within CVC) of those designs. Our heuristic can also be used with any decision procedure (not limited to CVC)

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

and any first-order theorem prover (not limited to Otter).

Refinement is a known method in the literature [20], [10], [14], where we start with a property and try to prove it. If that fails, the proof failure (the counterexample) is analyzed to provide guidance into how to refine the formula and try to prove it again. This process continues iteratively until the property is proved, disproved, or the available time or memory is exhausted. Our approach is different from other approaches in the literature in its use of predicate instantiation and the heuristics we use to find the refining predicates. That, as we show in this paper, makes our approach very simple yet very effective.

This paper is organized as follows. In section II-A, we define some terminology that we use in this paper. In section II-B, we review the idea of predicate instantiation. In section III-A, we describe a simple version of our refinement heuristic and then we show how it works on a simple example in section III-C. In section III-E, we describe the main heuristic which avoids some shortcomings of the first heuristic. We also show how this heuristic works on the same example. In section IV, we describe the experiments we did on a number of designs. Finally, we conclude with some remarks in section V.

II. Background

A. Terminology

Our goal in this paper is to check the validity of formulas of the form:

$$\forall \alpha^m \; \exists \beta^n \; [\Psi(\alpha^m, \beta^n)], \tag{1}$$

where we have m universally quantified variables $\alpha_0, ..., \alpha_{m-1}$ referred to by α^m and n existentially quantified variables $\beta_0, ..., \beta_{n-1}$ referred to by β^n. Ψ is a formula in the logic of equality and uninterpreted functions with linear arithmetic. Formula (1) falls into an undecidable class [3]. So, the refinement method that we introduce is a heuristic that is sound but not complete.

The main idea behind our refinement heuristics is to use counterexamples returned by CVC while checking the validity of the universal version of (1), which is $\forall \alpha^m, \beta^n \; [\Psi(\alpha^m, \beta^n)]$, to guide the formation of the instantiation predicate Φ (instantiation predicates will be described in the next section). A **counterexample** C of a formula Ψ is a consistent conjunction of literals, such that $C \models \neg\Psi$, where a literal is an atom (an equality, a predicate, or a Boolean variable) or its negation. We also refer to a conjunction of literals as a set whose elements are the literals in the conjunction. We also write $C(x_0, ..., x_l)$ to indicate the variables $x_0, ..., x_l$ that appear in the literals in a counterexample or a predicate C. When CVC checks the validity of a universal formula Ψ, it either returns `Valid` or returns a counterexample C, such that $C \models \neg\Psi$. We are interested in the smallest $C' \subseteq C$ such that $C' \models \neg\Psi$. To simplify the discussion, we assume that the counterexamples we get are minimized after they are returned by CVC. Minimizing a counterexample can be easily done by a simple greedy heuristic.

Given a counterexample $C(\alpha^m, \beta^n)$, we define $\mathbf{C}^\forall$ as the set of literals in C that contain only universally quantified variables (the αs) and $\mathbf{C}^\exists$ as the rest of the literals. For example, if C is$\{\alpha_0 = \alpha_1, \alpha_0 = \beta_0, P(\alpha_0, \alpha_1), P(\alpha_0, \beta_1), \beta_0 = f(\beta_1)\}$ then, $C^\forall$ is $\{\alpha_0 = \alpha_1, p(\alpha_0, \alpha_1)\}$ and $C^\exists$ is $\{\alpha_0 = \beta_0, p(\alpha_0, \beta_1), \; \beta_0 = f(\beta_1)\}$, where p is an uninterpreted predicate and f is an uninterpreted function.

B. Predicate Instantiation of Existential Quantifiers

The existential quantifier in a formula of the form:

$$\forall \alpha^m. \; \exists \beta^n. \; \Psi(\alpha^m, \beta^n) \tag{2}$$

is traditionally eliminated by constructing a set of functions $f_0(\alpha^m), f_1(\alpha^m), ..., f_{n-1}(\alpha^m)$ ($f^n(\alpha^m)$ for short) and replacing β_i by $f_i(\alpha^m)$:

$$\forall \alpha^m. \; \Psi(\alpha^m, \; f^n(\alpha^m)). \tag{3}$$

However, constructing such a function explicitly for hardware designs is a very tedious process. Instead, we apply a different technique which we call **predicate instantiation**. The idea is to find a predicate or a formula $\Phi(\alpha^m, \beta^n)$ such that the following two formulas can be proven valid:

$$\forall \alpha^m. \; \exists \beta^n. \; \Phi(\alpha^m, \beta^n) \tag{4}$$

$$\forall \alpha^m. \; \forall \beta^n. \; \Phi(\alpha^m, \beta^n) \Rightarrow \Psi(\alpha^m, \beta^n). \tag{5}$$

It is not hard to derive that the validity of (4) and (5) imply the validity of (2).

Note that to use the instantiation predicate, one still needs to prove an existential formula (4), which is undecidable in general. However, our experiments show that the resulting formulas are usually simple enough to be solved automatically by a theorem prover. The theorem prover we use is Otter [16] which takes a fraction of a second to prove each of the formulas we encountered.

In instantiation using functions as in (3), given a value α^m, the corresponding value of β^n is $f_1(\alpha^m), f_2(\alpha^m), ..., f_n(\alpha^m)$, where $f_i(\alpha^m)$ is the instantiation of β_i. In predicate instantiation, given an α^m, the set $\{\beta^n | \; \Phi(\alpha^m, \beta^n)\}$ is the set of values of β^n instantiated corresponding to the value α^m. We refer to this set by $\mathbf{Image}(\Phi, \alpha^m)$. We call Φ the **instantiation predicate**.

There are a number of advantages in using predicate instantiation. First, the formula (4) is usually very small as will be seen in the examples. That makes it possible to check its validity with a first order theorem prover without manual assistance. Second, the predicate Φ is usually constructed in a simple way from literals from the formula Ψ without the need for human assistance or intuition. This makes it possible to automate the process of finding Φ as will be shown in this paper.

In brief, our approach reduces the problem of checking the validity of a very large quantified formula into two much easier problems: 1. checking the validity of a similar formula without quantifiers using a tool like CVC, and 2. checking the validity of a very small quantified formula using a first-order theorem prover like Otter.

III. The Heuristic

For clarity of presentation, we present our heuristic in two steps. In section III-A, we start by describing Heuristic A, a simplified version of our final heuristic. In section III-B, we describe an example and then in section III-C, we show how Heuristic A works on this example. Then, we point out a limitation with this simplified heuristic in section III-D. In section III-E, we present our final heuristic, Heuristic B, which avoids the limitations of Heuristic A.

A. Heuristic A

Suppose we want to prove formula (1). We start by trying to prove the formula:

$$\forall \alpha^m, \beta^n \; [\Psi(\alpha^m, \beta^n)]. \tag{6}$$

We check the validity of (6) by CVC. If (6) is valid, then (1) is also valid and we are done. If (6) is not valid, CVC will return a counterexample C_1 such that:

$$\forall \alpha^m, \; \beta^n \; [C_1(\alpha^m, \beta^n) \Rightarrow \neg\Psi(\alpha^m, \beta^n)] \tag{7}$$

Formula (7) has the same form as formula (5), where C_1 here is the instantiation predicate. C_1 can be thought of as a "bad" *instantiation predicate* since it implies $\neg\Psi(\alpha^m, \beta^n)$ and causes the proof of (1) to fail. So, what we need is an instantiation predicate that avoids those "bad" values of βs instantiated by C_1. We use C_1 as a guide to find a better instantiation predicate Φ (we show how we do that later in this section) such that:

$$\forall \alpha^m \; \exists \beta^n \; [\Phi(\alpha^m, \beta^n)]. \tag{8}$$

This is required as explained in section II-B. Then we get a *refined* formula:

$$\forall \alpha^m, \; \beta^n \; [\Phi(\alpha^m, \beta^n) \Rightarrow \Psi(\alpha^m, \beta^n)] \tag{9}$$

and we use CVC again to try to prove this refined formula. If this formula is invalid, we repeat the refinement process iteratively. We stop when we reach a valid refined formula (9) or we cannot find an instantiation predicate that satisfies (8). This is shown in figure 1. In each iteration, the instantiation predicate, Φ, is updated with a new conjunct Φ' derived from the new counterexample returned by CVC. There can be a bound on the number of iterations of the main loop (line 2) or on the size of the instantiation predicate generated so far. If the limit is exceeded, the heuristic can just return the current counterexample as a potentially true counterexample. Computing Φ' is the critical part of this heuristic. We next describe two approaches to do that.

1) First Approach.: Since C_1 was a "bad" instantiation, we could avoid those values of β instantiated by C_1 as follows:

$$\forall \alpha^m, \; \beta^n \; [\beta^n \notin \mathit{Image}(C_1, \alpha^m) \Rightarrow \Psi(\alpha^m, \beta^n)], \tag{10}$$

where $\beta^n \notin \mathit{Image}(C_1, \alpha^m)$ is the instantiation predicate. This is equivalent to:

$$\forall \alpha^m, \; \beta^n \; [\neg C_1(\alpha^m, \beta^n) \Rightarrow \Psi(\alpha^m, \beta^n)]. \tag{11}$$

So, $\Phi(\alpha^m, \beta^n)$ here is $\neg C_1(\alpha^m, \beta^n)$. There are, however, several problems with this approach. 1. For any instantiation predicate candidate Φ, we need to prove that (8) is valid using a first order theorem prover. To be able to do that with a theorem prover, Φ must be a small and simple formula. However, counterexamples can be very complex, which can make it infeasible to check the validity of the formula above. 2. It will likely take several iterations before arriving at a Φ that satisfies (8) and (9). If in every iteration, we add a new conjunct, $\neg C_i$, to Φ, we will quickly get very large formulas in the form of conjunction of disjunctions. This can make it even harder for a first order theorem prover.

2) Second Approach.: To avoid the problems mentioned above, we use an alternative approach for finding Φ' which proved to be very effective in our experiments. Our approach is based on the observation that for all the hardware examples we considered, the instantiation predicates were simple conjunctions of literals. So, instead of adding the $\neg C_i$ to Φ in each iteration, we add $\neg \ell_j$, where ℓ_j is a literal in $C_i^{\exists}$ (see section II-A). We pick a literal from $C_i^{\exists}$ rather than $C_i^{\forall}$ because we are trying to instantiate the βs which do not appear in the literals in $C_i^{\forall}$. So, with this approach, lines 9 and 10 of Heuristic A would be:

```
 9. if there is any ℓ ∈ C^∃ s.t. ∀α^m ∃β^n [ Φ ∧ ¬ℓ ]
10.    then Φ ← Φ ∧ ¬ℓ
```

There are two main advantages in using literals instead of complete counterexamples in building Φ. 1. In each iteration, only one literal is added to Φ. This makes Φ much smaller and more tractable for the theorem prover. In addition, Φ remains a simple conjunction of literals compared to a conjunction of disjunctions in the other approach. This makes it easier to check its validity. 2. The conjunct $\neg \ell_j$ (where $\ell_j \in C_i$) is much stronger than the predicate $\neg C_i$ (i.e., $\neg \ell_j \Rightarrow \neg C_i$, but $\neg C_i \not\Rightarrow \neg \ell_j$). Stronger predicates result in stronger refinement in every iteration which in turn makes the heuristic converge much more quickly. Fewer iterations result in even smaller Φs. On the negative side, however, this stronger refinement can exclude "good" values of βs too. If this happens and we get stuck in later iterations of the heuristic, we can backtrack and change the literal we picked. The number of times the heuristic might need to backtrack is a function of the number of iterations and the number of possible choices in each iteration. As explained earlier and as experiments show, the number of iterations in our approach is very small. The number of choices in iteration i is bounded by the number of literals in $C_i^{\exists}$ which is also a small number in general. Hence, the number of times of backtracking is expected to be relatively small.

B. Example

In this section we apply Heuristic A to the 2-step induction formula needed to verify a simple read-only memory system that was used in [2]. This is a trivial design for the purpose of showing the basic ideas of the heuristic. The memory system

```
   Heuristic A (∀α^m ∃β^n[Ψ(α^m, β^n)])
 1. Φ ← true
 2. Repeat
 3. {
 4.  if ∀α^m, β^n[Φ(α^m, β^n) ⇒ Ψ(α^m, β^n)] is valid
 5.    then return (Valid)
 6.    else
 7.    {
 8.     let C be the counterexample
 9.     if there is a Φ' (derived from C)
                      s.t. ∀α^m ∃β^n [ Φ ∧ Φ' ]
10.       then Φ ← Φ ∧ Φ'
11.       else return (Failed)
12.    }
13. }
```

Fig. 1. Heuristic A

consists of a main memory with a one-line cache as shown in figure 2 (a). We call this memory system the *concrete* module and we verify it by proving that it is equivalent to an *abstract* memory system that represents the desired functionality. The abstract memory system is just a memory array as shown in figure 2 (b).

Let $N^c(s^c, \alpha)$ be the transition function for the concrete module, where s^c is the current state of the module and α is the input. Also, we use $R^c(s^c, \alpha)$ to refer to the result (output) returned by the concrete module when reading address α from the concrete memory in state s^c. Similarly, N^a and R^a refer to the transition function and output function respectively of the abstract module.

We prove the two modules equivalent by proving the formula:

$$\forall \ell, \alpha^\ell.\ R^c(N^c(s_0^c, \alpha^\ell)) = R^a(N^a(s_0^a, \alpha^\ell)), \qquad (12)$$

where, s_0^c and s_0^a are the initial states of the modules, $N(s_0, \alpha^\ell)$ is the state we get by running the module over an input sequence of length ℓ starting from state s_0, and $R(N(s_0, \alpha^\ell))$ is the last result returned by the module at the end of the run (this is a slight abuse of the notation to simplify the formula). Formula (12) says that starting both systems from the initial states (s_0^c and s_0^a) and running them on an arbitrary input sequence α^ℓ of length ℓ, both systems will generate the same sequence of outputs. This is the notion of *functional equivalence* [1] which implies that the concrete system is correct with respect to the abstract system.

The standard way to prove formula (12) is by induction on time:

$$\begin{array}{l} \forall \lambda.\ Q(s_0^c, s_0^a, \lambda) \\ \forall s^c, s^a, \sigma, \lambda\ \exists \beta.\ Q(s^c, s^a, \beta) \Rightarrow Q(N(s^c, \sigma), N(s^a, \sigma), \lambda), \end{array} \qquad (13)$$

where $Q(s^c, s^a, \lambda) \equiv R^c(s^c, \lambda) = R^a(s^a, \lambda)$. The premise of the induction step says that the two modules output the same value at state $\langle s^c, s^a \rangle$ (i.e., the two modules are equivalent in states s^c and s^a). In the consequent, we check if both systems output equal values after one transition (in other words, we check if the transition function preserves this equivalence relation).

This simple induction, however, fails most of the time due to incoherent (unreachable) states. For the example in figure 2, if we start from an incoherent state s^c where $a \neq b$, the induction step fails. When $\lambda = \pi$ and $\sigma \neq \pi$, the premise of the induction step simply says that $a = d$, but the consequent says that $b = d$. Clearly, the premise does not imply the consequent and the proof fails. In [2], it was shown that 2-step induction removes that incoherency problem:

$$\begin{array}{l} \forall s^c, s^a, \sigma^2, \lambda.\ \exists \alpha, \beta^2 \\ Q(s^c, s^a, \beta_0) \wedge Q(N(s^c, \alpha), N(s^a, \alpha), \beta_1) \Rightarrow \\ Q(N(s^c, \sigma^2), N(s^a, \sigma^2), \lambda). \end{array} \qquad (14)$$

To prove this formula with a validity checker, however, we need to eliminate the existential quantifiers. In [2], we showed that the instantiation predicate $\Phi \equiv \alpha \neq \pi \wedge \beta_0 = \beta_1 = \lambda$ enables us to prove the above formula using predicate instantiation:

$$\begin{array}{ll} \forall s^c, s^a, \sigma^2, \lambda.\ \exists \alpha, \beta^2. & \Phi \\ \forall s^c, s^a, \sigma^2, \lambda, \alpha, \beta^2. & \Phi \Rightarrow \Psi, \end{array} \qquad (15)$$

where Ψ is the formula in (14) without the quantifiers. The idea behind using 2-step induction with this instantiation predicate is as follows. The premise of the induction step in (15) (Φ conjoined with the premise of Ψ) corresponds to reading from λ in the concrete memory, then flushing the cache by reading from $\alpha \neq \pi$, then reading from λ again and asserting that the two reads are equal. When $\lambda = \pi$ (which is the case that uncovered the incoherency in the simple induction proof), the premise basically says that $a = b$ which is enough to guarantee that the concrete system is coherent and hence the induction proof goes through.

In [2], we used reasoning similar to the paragraph above to find this instantiation predicate. The designer was expected to use his intuition to find those predicates. In this section, we show how Heuristic A can find this instantiation predicate completely automatically.

C. Applying Heuristic A

The 2-step induction formula we need to prove is formula (14). In this formula, we have 3 existentially quantified variables (α, β_0, and β_1) and 5 universally quantified variables (s^c, s^a, λ, σ_0, and σ_1). We start applying the heuristic with a $\Phi \equiv true$. The first step is to use CVC to check the validity of the formula $\Phi \Rightarrow \Psi$, where Ψ is the formula (14) without the quantifiers. This formula is invalid and CVC returns the counterexample:

$$C \equiv \beta_0 \neq \lambda \wedge \lambda = \pi \wedge \sigma_1 \neq \pi \wedge a \neq b. \qquad (16)$$

The variables π, a, and b are part of the state s^c and hence are universally quantified. Now, we find $C^\forall$ and $C^\exists$ to be:

$$\begin{array}{ll} C^\forall & \equiv \{\sigma_1 \neq \pi, a \neq b, \lambda = \pi\} \\ C^\exists & \equiv \{\beta_0 \neq \lambda\} \end{array}$$

As we can see, $C^\exists$ has only one literal in it and hence we don't have a choice of which literals to use for adding to Φ. So,

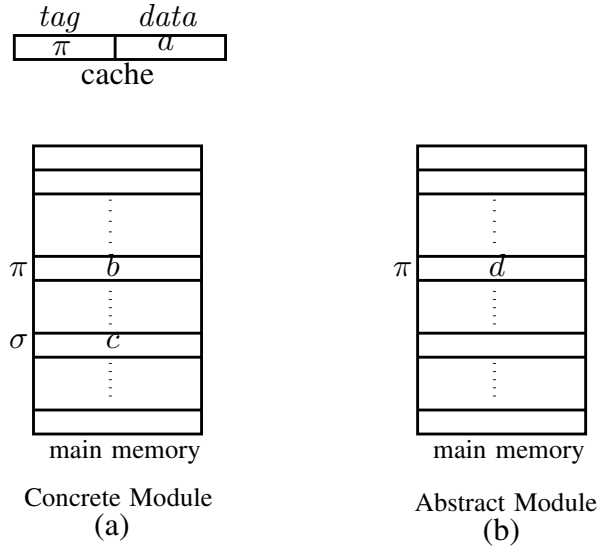


Fig. 2. Memory Example

Φ becomes $\beta_0 = \lambda$. We check $\forall\lambda.\,\exists\beta_0.\,[\beta_0 = \lambda]$ using Otter and we find it valid. Next, we check the validity of $\Phi \Rightarrow \Psi$ using CVC and we get back a counterexample $C \equiv \beta_1 \neq \lambda \wedge \lambda = \pi \wedge \sigma_1 \neq \pi$. Again, we find $C^{\exists} \equiv \{\beta_1 \neq \lambda\}$. And once more, we are left with only one option in $C^{\exists}$ and we update Φ to be $\beta_0 = \lambda \wedge \beta_1 = \lambda$ and we check it with Otter. Now, we check the validity the updated $\Phi \Rightarrow \Psi$ with CVC. The formula is still invalid and we get the counterexample $C \equiv \alpha = \pi \wedge \lambda = \pi \wedge \sigma_1 \neq \pi$ and $C^{\exists} \equiv \{\alpha = \pi\}$. So, we update Φ to be $\beta_0 = \lambda \wedge \beta_1 = \lambda \wedge \alpha \neq \pi$ and we check it with Otter. Now, we use CVC to check the validity of the formula $\Phi \Rightarrow \Psi$ and this time CVC asserts the validity of this formula. With this, we have proved that (14) is valid and hence proved the correctness of the memory system completely automatically. Notice that the instantiation predicate Φ found by the heuristic is exactly like the one we found manually.

D. Limitations of Heuristic A

Heuristic A is highly sensitive to the form of the counterexample returned by CVC. If the same counterexample is returned in a different shape, the heuristic might fail in finding the right instantiations. Consider for example the one-line cache. The first counterexample we got back from CVC was $C \equiv \beta_0 \neq \lambda \wedge \lambda = \pi \wedge \sigma_1 \neq \pi \wedge a \neq b$ and $C^{\exists} \equiv \{\beta_0 \neq \lambda\}$. From this counterexample, Heuristic A extracted the first conjunct in the instantiation vector, $\beta_0 = \lambda$ which was a good decision to make. Now, assume that CVC returned a logically equivalent, but syntactically different counterexample $C' \equiv \beta_0 \neq \pi \wedge \lambda = \pi \wedge \sigma_1 \neq \pi \wedge a \neq b$. Looking at this counterexample, it is easy to see that it is equivalent to C. The difference here is that instead of $\beta_0 \neq \lambda$, we have $\beta_0 \neq \pi$, but these two literals are equivalent in the context of these counterexamples because $\lambda = \pi$. According to Heuristic A, $C'^{\exists} \equiv \{\beta_0 \neq \pi\}$. Next, Heuristic A would pick $\beta_0 = \pi$ as the new literal to add to Φ. But it is clear that this literal will not help and the heuristic will fail in constructing a useful instantiation predicate. The same problem will happen with later iterations of the heuristic.

E. Heuristic B

In this section, we describe a technique to deal with the limitation described in the previous section. In the example we described above, Heuristic A picked $\beta_0 = \pi$ to add to Φ, which fails to be a useful instantiation. This instantiation, however, is still good when $\pi = \lambda$. In other words, it is still a good instantiation within the *context* of this counterexample. The *context* of a counterexample is the subspace of values of α^m that satisfy the counterexample. Within the context of a counterexample, the shape of the counterexample does not matter. For the example above, within the context of C', $\beta_0 = \pi$ and $\beta_0 = \lambda$ are equivalent, because C' implies that $\beta_0 = \lambda$. The main idea in Heuristic B is to divide the space of α^m into a number of subspaces based on the contexts of counterexamples returned by CVC. Then, for each subspace, an instantiation predicate is constructed. Dividing the space based on the contexts of counterexamples and finding a different instantiation for each subspace, makes the heuristic more robust against the syntactic form of the counterexample, which avoids the problem described in the previous section.

Formally, we need to find a number of predicates $\Pi_0(\alpha^m), \Pi_1(\alpha^m), ..., \Pi_j(\alpha^m)$ such that for $0 \leq i \leq j$:

$$\begin{array}{ll} \forall\alpha^m.\ \exists\beta^n. & \Phi_i(\alpha^m, \beta^n) \\ \forall\alpha^m,\ \beta^n. & \Phi_i(\alpha^m, \beta^n) \wedge \Pi_i(\alpha^m) \Rightarrow \Psi(\alpha^m, \beta^n) \end{array} \tag{17}$$

and

$$\forall\alpha^m.\ \Pi_0(\alpha^m) \vee \Pi_1(\alpha^m) \vee ... \vee \Pi_j(\alpha^m). \tag{18}$$

where $\Pi_i(\alpha^m)$ is a predicate that defines the i^{th} subspace. For the cache example, we can first consider the formula $\pi = \lambda \Rightarrow \Psi$ and find the right instantiation for it. Then, we consider the formula $\pi \neq \lambda \Rightarrow \Psi$ and find its instantiation which could be different from the first one. In other words, for this example, we have two subspaces: $\Pi_0 \equiv \pi = \lambda$ and $\Pi_1 \equiv \pi \neq \lambda$. It follows from (17) and (18) that $\forall\alpha^m.\ \exists\beta^n.\ \Psi(\alpha^m, \beta^n)$. This is shown in Appendix A.

Heuristic B uses heuristic `Check` (shown in figure 3) to divide the α^m space into subspaces and find the appropriate instantiation predicate for each subspace. Heuristic `Check` takes as input a formula $\forall\alpha^m.\ \exists\beta^n.\ \Psi(\alpha^m, \beta^n)$ and returns a subspace predicate $\Pi(\alpha^m)$ such that:

$$\forall\alpha^m\ [\Pi(\alpha^m) \Rightarrow \exists\beta^n\ \Psi(\alpha^m, \beta^n)]. \tag{19}$$

In each iteration of `Check`, $C^{\forall}$ is used as the context to refine the subspace of α^m (line 13). We avoid refining the α^m space by adding any literals from $C^{\exists}$. Keeping Π a predicate of α^m (and not β^n) is needed to guarantee the soundness of our heuristic as shown in the Appendix. $C^{\forall}$is a simple yet good approximation of the context of C.

If `Check` cannot find such Π, it returns `Failed`. If `Check` is successful in finding such a Π, it returns it after *minimizing* it by a call to the function `Minimize`. This function finds a smallest subset, Π', of the literals in Π such that (19) still

```
   Check (∀α^m ∃β^n[Ψ(α^m, β^n)])
 1. Φ ← true
 2. Π ← true
 3. Repeat
 4. {
 5.  if ∀α^m, β^n [Π(α^m) ∧ Φ(α^m, β^n)
          ⇒ Ψ(α^m, β^n)] is valid
 6.    then Minimize(Π)
 7.         return (Valid, Π)
 8.    else
 9.     {
10.      let C be the counterexample
11.      if there is any ℓ ∈ C^∃ s.t.
                 ∀α^m ∃β^n [ Φ ∧ ¬ℓ ]
12.        then Φ ← Φ ∧ ¬ℓ
13.             Π ← Π ∧ C^∀
14.        else return (Failed)
15.     }
16. }
```

Fig. 3. Heuristic Check

holds. This could be a greedy heuristic that drops one literal from Π at a time and checks that (19) still holds until it finds the smallest such Π. The reason Π is minimized is to find the largest subspace that is covered by the instantiation predicate Φ; the less literals in Π the larger the subspace it represents.

The main heuristic, Heuristic B, is shown in figure 4. It starts with $\Sigma \equiv true$, where Σ represents the subspace that was not covered in previous iterations. In each iteration, Heuristic `Check` is called with the formula:

$$\forall \alpha^m,\ \exists \beta^n\ [\Sigma(\alpha^m) \Rightarrow \Psi(\alpha^m, \beta^n)].$$

If `Check` returns `Failed`, Heuristic B also returns `Failed`. If `Check` returns `Valid`, it also returns the subspace that was checked σ (i.e., the subspace σ for which `Check` proved that $\forall \alpha^m.\ \sigma(\alpha^m) \wedge \Sigma(\alpha^m) \Rightarrow \exists \beta^n\ \Psi(\alpha^m, \beta^n)$ is valid). So, we add $\neg\sigma$ to Σ indicating that subspace σ was checked and we do not need to consider it in later iterations. This keeps repeating until we exit as in line 11 or we reach an iteration where:

$$\forall \alpha^m,\ \beta^n\ [\Sigma(\alpha^m) \Rightarrow \Psi(\alpha^m, \beta^n)] \tag{20}$$

is valid and we return `Valid` (line 5). The proof of soundness of this heuristic can be found in Appendix A.

Heuristic A was limited to formulas that can be proved with instantiation predicates that are conjunctions of literals. Heuristic B, however, can handle a much larger class of formulas by splitting the space of models into several subspaces and finding a different instantiation predicate for each subspace. This makes it possible to handle a larger class of formulas while maintaining simple conjunctive instantiation predicates that can be easily handled by first-order theorem provers.

F. Example

In this section we consider the one-line cache example again and apply Heuristic B to it. Below, we show the heuristic

```
   Heuristic B (∀α^m ∃β^n[Ψ(α^m, β^n)])
 1. Σ ← true
 2. Repeat
 3.  {
 4.   if ∀α^m, β^n [Σ(α^m) ⇒ Ψ(α^m, β^n)] is Valid
 5.     then return (Valid)
 6.     else
 7.      {
 8.       (result, σ) ←
            Check (∀α^m ∃β^n[Σ(α^m) ⇒ Ψ(α^m, β^n)])
 9.       if result = Valid
10.         then Σ ← Σ ∧ ¬σ
11.         else return (Failed)
12.      }
13.  }
```

Fig. 4. Heuristic B

through the different iterations.

- $\Sigma \equiv true$.
- **Iteration 1** (Heuristic B)
  - $\Sigma \Rightarrow \Psi$ is invalid.
  - call `Check` $(\Sigma \Rightarrow \Psi)$
    - **Iteration 1** (Heuristic `Check`)
      - $\Pi \wedge \Phi \Rightarrow \Psi$ is invalid.
      - $C \equiv \beta_0 \neq \pi \wedge \lambda = \pi \wedge \sigma_1 = \pi \wedge a \neq b,\ C^{\exists} \equiv \{\beta_0 \neq \pi\}$.
      - $\Pi \equiv \lambda = \pi \wedge \sigma_1 = \pi,\ \Phi \equiv \beta_0 = \pi$.
    - **Iteration 2** (Heuristic `Check`)
      - $\Pi \wedge \Phi \Rightarrow \Psi$ is invalid.
      - $C \equiv \beta_1 \neq \pi \wedge \lambda = \pi \wedge \sigma_1 = \pi \wedge a \neq b,\ C^{\exists} \equiv \{\beta_1 \neq \pi\}$.
      - $\Pi \equiv \lambda = \pi \wedge \sigma_1 = \pi,\ \Phi \equiv \beta_0 = \pi \wedge \beta_1 = \pi$.
    - **Iteration 3** (Heuristic `Check`)
      - $\Pi \wedge \Phi \Rightarrow \Psi$ is invalid.
      - $C \equiv \alpha = \lambda \wedge \lambda = \pi \wedge \sigma_1 = \pi \wedge a \neq b,\ C^{\exists} \equiv \{\alpha = \lambda\}$.
      - $\Pi \equiv \lambda = \pi \wedge \sigma_1 = \pi,\ \Phi \equiv \beta_0 = \pi \wedge \beta_1 = \pi$.
    - **Iteration 4** (Heuristic `Check`)
      - $\Pi \wedge \Phi \Rightarrow \Psi$ is valid.
      - `Minimize` Π.
      - Return $\Pi \equiv \lambda = \pi$.
  - $\sigma \equiv \lambda = \pi$.
  - $\Sigma \equiv \lambda \neq \pi$.
- **Iteration 2** (Heuristic B)
  - $\Sigma \Rightarrow \Psi$ is invalid.
  - call `Check` $(\Sigma \Rightarrow \Psi)$
    - **Iteration 1** (Heuristic `Check`)
      - $\Pi \wedge \Phi \Rightarrow \Psi$ is invalid.
      - $C \equiv \lambda \neq \pi \wedge \equiv \beta_0 \neq \lambda \wedge \sigma_1 = \pi \wedge a \neq b,\ C^{\exists} \equiv \{\beta_0 \neq \lambda\}$.
      - $\Pi \equiv \lambda \neq \pi \wedge \sigma_1 = \pi,\ \Phi \equiv \beta_0 = \lambda$.
    - **Iteration 2** (Heuristic `Check`)
      - $\Pi \wedge \Phi \Rightarrow \Psi$ is invalid.
      - $C \equiv \lambda \neq \pi \wedge \equiv \beta_1 \neq \lambda \wedge \sigma_1 = \pi \wedge a \neq b,\ C^{\exists} \equiv \{\beta_1 \neq \lambda\}$.

· $\Pi \equiv \lambda \neq \pi \wedge \sigma_1 = \pi$, $\Phi \equiv \beta_0 = \lambda \wedge \beta_1 = \lambda$.

∗ **Iteration 3** (Heuristic `Check`)

· $\Pi \wedge \Phi \Rightarrow \Psi$ is invalid.

· $C \equiv \lambda \neq \pi \wedge \equiv \alpha = \pi \wedge \sigma_1 = \pi \wedge a \neq b$, $C^{\exists} \equiv \{\alpha = \pi\}$.

· $\Pi \equiv \lambda \neq \pi \wedge \sigma_1 = \pi$, $\Phi \equiv \beta_0 = \lambda \wedge \beta_1 = \lambda \wedge \alpha \neq \pi$.

∗ **Iteration 4** (Heuristic `Check`)

· $\Pi \wedge \Phi \Rightarrow \Psi$ is valid.

· `Minimize` Π.

· Return $\Pi \equiv true$.

– $\sigma \equiv true$.

– $\Sigma \equiv false$.

• **Iteration 3** (Heuristic B)

– $\Sigma \Rightarrow \Psi$ is valid.

– Return (`Valid`).

IV. Experiments

In addition to the one-line cache example, we used this heuristic to verify four more examples: (1) unbounded direct-mapped cache, (2) unbounded two-level direct mapped cache, (3) unbounded 2-way set-associative cache, and (4) simple DLX-like pipeline. These designs are available at http://verify.stanford.edu/refinement. In all of these examples, the heuristic was able to find the right instantiation predicates in a small number of iterations. Processing and passing the formulas to and from CVC and Otter was done manually.

Some statistics about the designs are shown in figure 5. The second column shows the number of nodes in the formula we checked for each design. These formulas were generated by k-step induction and symbolic simulation. The nodes in the formulas include word-level *ite*'s, Boolean operators, linear arithmetic operators, uninterpreted functions and predicates, equalities, unbounded array reads, and unbounded array writes. The formulas are in structural form.

The remaining columns show the number of unbounded integer variables (I), the number of Boolean variables (B), the number of finite-domain variables (F), and the number of unbounded arrays (A) in each design. Some of the arrays were over integers, while others were over complex tuples of data.

Checking the validity of each of these formulas results in several calls to Otter and CVC. The calls to Otter were with very small formulas (the instantiation predicates); each call took less than a second (on a Pentium 3 800MHz machine). Hence, the time spent on proving quantified formulas was a small fraction.

Each call to CVC is on a slightly modified version of the original formula (the k-step induction formula without the quantifiers) with only a few literals added as described in the heuristic. Our heuristic changes the formula in a minor way that does not change the CVC run time per call.

So, from a practical point of view, the heuristic reduced the problem of checking the validity of large quantified formulas into the problem of validity checking of a number of quantifier-free formulas of approximately the same size as the original formula.

	formula size	I	B	F	A
direct mapped cache	9450	35	6	3	3
2-level direct mapped cache	31860	44	9	4	4
2-way set associative cache	17289	36	11	3	3
DLX pipeline	2862	50	6	5	4

Fig. 5. Verified Designs

V. Conclusions

We introduced a refinement heuristic for checking the validity of first-order formulas of the form $\forall\alpha^m\ \exists\beta^n.\ \Psi(\alpha^m, \beta^n)$. The use of predicate instantiation along with our heuristic for finding instantiation predicates proved to be very effective in proving complex quantified formulas from hardware verification.

Our approach reduces the problem of checking the validity of large quantified formulas into two much easier problems: 1. checking the validity of a number of similar formulas that are quantifier-free with a validity checker like CVC and 2. checking the validity of a number of very small quantified formulas with a first-order theorem prover like Otter.

Experiments show the effectiveness of our approach in hardware verification as well as theorem proving.

Acknowledgment

The authors would like to acknowledge the National Science Foundation CCR-0121403, and KFUPM, Saudi Arabia for supporting this research. The content of this paper does not necessarily reflect the position or the policy of NSF, or KFUPM, and no official endorsement should be inferred.

The authors would also like to thank the anonymous reviewers for the helpful comments on the paper.

References

[1] Husam Abu-Haimed. *Automatic Generation of Invariants in Formal Verification of Microprocessors and Memory Systems*. PhD thesis, Electrical Engineering Department, Stanford University, 2004.

[2] Husam Abu-Haimed, Sergey Berezin, and David L. Dill. Strengthening Invariants by Symbolic Consistency Testing. In *CAV*, 2003.

[3] W. Ackermann. *Solvable Cases of the Decision Problem*. North-Holland Publishing Company, 1954.

[4] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful Techniques for the Automatic Generation of Invariants. In *CAV*, 1996.

[5] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic Generation of Invariants and Intermediate Assertions. In *Theoretical Computer Science*, 1997.

[6] Jerry R. Burch. Techniques for Verifying Superscalar Microprocessors. In *Design Automation Conference (DAC)*, 1996.

[7] Jerry R. Burch and David L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *CAV*, 1994.

[8] Michael Colon and Tomas E. Uribe. Generating Finite-State Abstractions of Reactive Systems Using Decision Procedures. In *CAV*, 1998.

[9] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective Theorem Proving for Hardware Verification. In *International Conference on Theorem Provers in Circuit Design*, 1994.

[10] Satyaki Das and David L. Dill. Counter-Example Based Predicate Discovery in Predicate Abstraction. In *FMCAD*, 2002.

[11] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with Predicate Abstraction. In *CAV*, 1999.

[12] S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *CAV*, 1997.

[13] Warren A. Hunt. A Verified Microprocessor. In *Lecture Notes in Computer Science*, volume 795. Springer-Verlag, 1994.
[14] Andrew Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal od Automated Reasoning*, 16, 1996.
[15] R. B. Jones, J. Skakkebæk, and D. L. Dill. Reducing Manual Effort in Abstraction of Out-Of-Order Execution. In *FMCAD*, 1998.
[16] William W. McCune. Otter 3.0 Reference Manual and Guide. Technical report, ANL, 1994.
[17] Ken McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. In *CAV*, 1998.
[18] Ken McMillan. Verification of Infinite State Systems by Compositional Model Checking. In *CHARME*, 1999.
[19] Tom Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, 1993.
[20] Jamie Stark and Andrew Ireland. Invariant Discovery via Failed Proof Attempts. In *International Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR)*, 1998.
[21] A. Stump, C. Barrett, and D. L. Dill. CVC: a Cooperating Validity Checker. In *CAV*, 2002.
[22] Jeffrey X. Su, David L. Dill, and Clark W. Barrett. Automatic Generation of Invariants in Processor Verification. In *FMCAD*, 1996.
[23] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A Technique for Invariant Generation. In *TACAS*, 2001.
[24] Miroslav N. Velev and Randal E. Bryant. Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction. In *Design Automation Conference (DAC)*, 2000.

APPENDIX A
SOUNDNESS OF HEURISTIC B

We show that if Heuristic B returns `Valid`, then the formula $\forall\alpha^m\ \exists\beta^n\ \Psi(\alpha^m,\beta^n)$ is valid. Let σ_i be the σ returned in iteration i and let Σ_i be the Σ at the beginning of iteration i. Hence, $\Sigma_0 \equiv true$ and $\Sigma_i \equiv \Sigma_{i-1} \wedge \neg\sigma_{i-1}$.

Lemma 1: If a call to heuristic `Check` in iteration i returns `Valid` with a predicate σ_i, then:

$$\forall\alpha^m\ [\sigma_i(\alpha^m)\wedge\neg\sigma_{i-1}(\alpha^m)\wedge...\wedge\neg\sigma_0(\alpha^m) \Rightarrow \exists\beta^n\ \Psi(\alpha^m,\beta^n)].$$

Proof: The first thing to note is that if `Check` returns `Valid` with a predicate σ_i, then there is a $\Phi(\alpha^m,\beta^n)$ such that:

$$\forall\alpha^m\ \exists\beta^n\ [\Phi(\alpha^m,\beta^n)]. \tag{21}$$

$$\forall\alpha^m,\beta^n\ [\sigma_i(\alpha^m) \wedge \Sigma_i(\alpha^m) \wedge \Phi(\alpha^m,\beta^n) \Rightarrow \Psi(\alpha^m,\beta^n)]. \tag{22}$$

From (21) and (23) we conclude:

$$\forall\alpha^m\ \exists\beta^n\ [\sigma_i(\alpha^m) \wedge \Sigma_i(\alpha^m) \Rightarrow \Psi(\alpha^m,\beta^n)], \tag{23}$$

which is equivalent to:

$$\forall\alpha^m\ [\sigma_i(\alpha^m) \wedge \Sigma_i(\alpha^m) \Rightarrow \exists\beta^n\ \Psi(\alpha^m,\beta^n)], \tag{24}$$

which can be rewritten as:

$$\forall\alpha^m\ [\sigma_i(\alpha^m)\wedge\neg\sigma_{i-1}(\alpha^m)\wedge...\wedge\neg\sigma_0(\alpha^m) \Rightarrow \exists\beta^n\ \Psi(\alpha^m,\beta^n)]. \tag{25}$$

■

Lemma 2: If a call to Heuristic B with $\forall\alpha^m\ \exists\beta^n\ \Psi(\alpha^m,\beta^n)$ returns `Valid`, then the formula $\forall\alpha^m\ \exists\beta^n\ \Psi(\alpha^m,\beta^n)$ is valid.

Proof: Suppose Heuristic B returns `Valid` in the k^{th} iteration. In this case, we know that:

$$\forall\alpha^m,\beta^n\ [\Sigma_k(\alpha^m) \Rightarrow \Psi(\alpha^m,\beta^n)] \tag{26}$$

Formula (26) implies:

$$\forall\alpha^m.\ [\Sigma_k(\alpha^m) \Rightarrow \exists\beta^n.\ \Psi(\alpha^m,\beta^n)] \tag{27}$$

Since the heuristic returned `Valid`, by Lemma 1, we know that:

$$\forall\alpha^m.\ [\,\sigma_i(\alpha^m)\wedge\neg\sigma_{i-1}(\alpha^m)\wedge...\wedge\neg\sigma_0(\alpha^m) \Rightarrow \exists\beta^n\ \Psi(\alpha^m,\beta^n)\,] \tag{28}$$

is valid, for $0 \le i < k$.

Now, consider the case where Σ_k is false. If Σ_k is false, it means there must be at least one i such that σ_i is true. From this and formula (28), we conclude that whenever Σ_k is false, then:

$$\forall\alpha^m\ \exists\beta^n.\ [\Psi(\alpha^m,\beta^n)].$$

In other words, we know that:

$$\forall\alpha^m.\ [\,\neg\Sigma_k(\alpha^m) \Rightarrow \exists\beta^n\Psi(\alpha^m,\beta^n)\,]. \tag{29}$$

From formula (27) and formula (29), we conclude:

$$\forall\alpha^m\ \exists\beta^n.\ [\,\Psi(\alpha^m,\beta^n)\,], \tag{30}$$

■

Notice that in deriving formulas (27) and (29), we used the fact that β^n does not appear in Σ_k and hence we were able to push in the $\exists\beta^n$ quantifier. If Σ has any literals containing β^n, then a `Valid` returned by Heuristic B does not imply the validity of $\forall\alpha^m\ \exists\beta^n.\ [\,\Psi(\alpha^m,\beta^n)\,]$. That is why we do not add any literals from $C^{\exists}$ to Π and hence to Σ.

An Integration of HOL and ACL2

Michael J.C. Gordon, James Reynolds
University of Cambridge Computer Laboratory
William Gates Building, 15 JJ Thomson Avenue
Cambridge CB3 0FD, United Kingdom
mjcg@cl.cam.ac.uk, jr291@cam.ac.uk

Warren A. Hunt, Jr., Matt Kaufmann
Department of Computer Sciences
1 University Station, M/S C0500
Austin TX 78712-0233, USA
hunt@cs.utexas.edu, kaufmann@cs.utexas.edu

Abstract

A link between the ACL2 and HOL4 proof assistants is described. This allows each system to be deployed smoothly within a single formal development. Several applications are being considered: using ACL2's execution environment for simulating HOL models; using ACL2's proof automation to discharge HOL proof obligations; and using HOL to specify and verify properties of ACL2 functions that cannot easily be stated in the first-order ACL2 logic.

Care has been taken to ensure sound translations between the logics supported by HOL and ACL2. The initial ACL2 theory has been defined inside HOL, so that it is possible to prove mechanically that first-order ACL2 functions on S-expressions correspond to higher-order functions operating on a variety of types. The translation between the two systems operates at the level of S-expressions and is intended to handle large hardware and software models.

1. Introduction

Higher-Order Logic (HOL) and First-Order Logic (FOL) are both used to model hardware and software. Separate verification communities have evolved based on each kind of logic (e.g. a verification group in one major processor company uses higher-order logic, whilst the corresponding group in a competitor company uses first-order logic). There are projects in progress that use models in both logics (e.g. the Cryptol/AAMP7 project at Rockwell Collins and Galois, Inc., which is discussed briefly later). In this paper we describe a method of linking the HOL4 [21] and ACL2 [16] proof assistants in a way that enables the strengths of each system to be smoothly deployed within a single formal development and the resulting verifications to be evaluated to very high levels of assurance. Our work links two particular systems, but the approach is intended to be portable: we are investigating linking Isabelle/HOL [20] and ACL2.

The key idea is a HOL theory, SEXP, that bridges the gap between the HOL4 and ACL2 logics. HOL4 developments are mapped to SEXP developments and the relationship verified by proof. SEXP developments closely correspond to ACL2 developments and can be converted to them by simple reading and writing of files containing S-expressions.

Our initial motivation was to use ACL2 for high performance simulation of HOL models (see Section 7). A different kind of application is to use higher-order logic to specify properties that cannot easily be stated in the first-order ACL2 logic. An instance of this is validating the translation of Cryptol [4] programs to AAMP7 [26] binary code whose semantics are defined in ACL2, as currently being undertaken by Galois, Inc. and Rockwell Collins [10, 23]. The idea is to start with a Cryptol program, translate it to an ACL2 function definition, import the generated ACL2 definition into HOL and then prove, using the semantics of Cryptol, which is formulated in higher-order logic, that the translated Cryptol program correctly implements the source specification. Another example, which we are working on in collaboration with Joe Hurd, is to verify that an ACL2 function implementing a probabilistic primality test satisfies a specification based on concepts from measure theory [13]. ACL2 executes this algorithm very fast, but it is hard to express its specification in the ACL2 logic.

In the next section we discuss previous work on connecting HOL and Boyer-Moore provers. The underlying logical ideas for embedding ACL2 in HOL are then motivated and the theory of the ACL2 logic in higher order logic is described. We then give details, using simple examples, of how we convert between HOL and ACL2. Next comes a simple example illustrating the use of ACL2 to execute HOL, together with runtime data illustrating the kind of performance gains that can be achieved. The paper ends with future work and some conclusions.

We use "HOL" both to refer to higher-order logic and to proof systems supporting the logic. We use "HOL4" when that particular implementation is intended. We use "ACL2" for both the system and the logic it supports.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

2. Related work

In 1991, Fink et al. described a proof manager PM [6] that enabled HOL input to be transformed into "first-order assertions suited to the Boyer-Moore prover". In 1999 Mark Staples implemented a tool called ACL2PII for linking ACL2 and HOL98 [28]. As far as we know these are the only previous attempts to link HOL to ACL2. ACL2PII was used by Susanto and Melham [29, 30].

Both PM and ACL2PII translate between higher-order logic and first-order logic. When translating from untyped Boyer-Moore logic to typed higher-order logic it can be hard to figure out which types to assign. Staples points out that the ACL2 S-expression `NIL` might need to be translated to `F` (boolean type), or `[]` (list type) or `NONE` (option type), depending on context. The ACL2PII user has to set up "translation specifications" that are pattern-matching rewrite rules to perform the ACL2-to-HOL translation. These are encoded in ML and are thus not supported by any formal validation.

The previous links between HOL and Boyer-Moore/ACL2 systems have been open to the criticism that the translation of formulae between the systems may be unsound. Our contribution is to design and implement an approach that provides high assurance that the corresponding HOL and ACL2 formulae have equivalent semantics. Our approach provides higher assurance than that provided by PM or ACL2PII because we perform *proof-based formal translation* between the higher-order logic formulae of HOL4 and the first-order formulae of ACL2. The logically tricky parts of the translations are done within a formal framework, namely translation to SEXP within HOL. Thus the meta-language scripts of PM or ACL2PII are replaced by deductions in the HOL4 system together with a clean and semantically simple link between SEXP and ACL2.

There has been lots of work on connecting together other proof assistants. Felty and Howe [5] import HOL90 theories into Nuprl. The theory justifying this is sophisticated (the two logics are fairly different) and the link is only one way. More recently, Mason and Talcott linked the Maude rewriting system to PVS [18] using an architecture based on the Actor model of computation. Both of these linkings use complex meta-theory to justify the soundness of transferring formulae between logics. In contrast, our approach mechanically checks the semantically tricky parts of the linkage (higher-order logic to SEXP). The actual data transfer between HOL4 and ACL2 uses terms (S-expressions) with identical structure and meanings in each system.

Recently, motivated by the Flyspeck project [31], tools have been implemented [19] to move Isabelle/HOL developments into HOL Light [11] and HOL Light developments into Isabelle/HOL [22]. HOL light is a proof assistant for exactly the same logic as the one supported by HOL4, but it has slightly different proof infrastructure and is implemented in O'Caml rather than Standard ML. Isabelle/HOL [20] is a proof assistant for a significantly more complex and expressive version of higher order logic than the one supported by HOL4 and HOL Light (e.g. it has type classes). Obua and Skalberg's paper provides a framework for importing HOL4 and HOL Light proofs into the Isabelle/HOL logic and then replaying them inside the Isabelle/HOL system. No trusting of the source system (HOL4 or HOL Light) is needed. A proof recording mechanism generates proof scripts that can be replayed. This direction is logically easy as the logic being imported is a subset of the Isabelle/HOL logic. The paper by McLaughlin describes a transfer of proofs going the other way (Isabelle/HOL to HOL Light) and shows how to generate ML code for Isabelle/HOL type classes than can replay class instance proofs inside HOL Light, using ML functors to elaborate type class instances. The work in these two papers bridges a smaller semantic gap than that from HOL to ACL2, but accomplishes more, namely the translation of proofs as well as formulae. HOL4, HOL Light and Isabelle/HOL are all LCF-style tactic-based systems, so share the same proof scripting methodology. The proof development methodologies of HOL and ACL2 are not at all the same, so harder to link. Developing tools to mechanise the replaying of HOL proofs in ACL2, and vice versa is a topic for future research.

The linking of proof assistants to automatic tools, like decision procedures, automatic theorem provers and model checkers, has been studied extensively. One approach is to embed the logic of the automatic tool in the logic of the proof assistant. This may be easy as the former logic might just be a subset of the latter logic. For example, SAT solvers operate on propositional logic, which is a subset of first order logic or higher order logic. However, in some cases the automatic tool has a specialised logic. Examples of this are model checkers, which decide properties of Kripke structures (the models) expressed in temporal logic. One approach to linking these is to build a theory representing the formulae of the specialised logic in the proof assistant logic, which is usually some kind of higher order logic. The languages used for model checking (Kripke structure notations for models and temporal logic for properties) can easily be defined in higher order logic. Problems solvable by model checking can then be converted (e.g. by rewriting) into formulae in the embedded checkable language, and then exported to external model checkers, which are used as trusted oracles. An early pioneering example linked PVS to a symbolic mu-calculus checker [24] and more recently SMV has been linked to HOL4 [32]. This approach to linking model checkers to proof assistants is similar to the way we are linking HOL4 and ACL2. However, the details and general flavour are different since the logics of HOL4 and ACL2

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

are both general purpose specification languages. It is easy to define Kripke structures and temporal logic operators in higher order logic, but it turns out to be quite complex and tricky to embed the entire ACL2 logic in HOL.

3. ACL2: axiomatic theory or interpreter?

Consider the ACL2 axiom `ASSOCIATIVITY-OF-*` occurring in the ACL2 source file `axioms.lisp`: `(EQUAL (* (* X Y) Z) (* X (* Y Z)))`. This can be viewed as an S-expression in ACL2's version of Lisp, or as a formula of first-order logic.

Under the first view the axiom is valid because if `X`, `Y` and `Z` are replaced by any S-expressions, then the resulting instance of the axiom will evaluate to 'true', i.e., `T` in Common Lisp. Under the second view, the formula is an axiom that defines what it means for evaluation to be correct: it is a partial semantics of Lisp evaluation. Thus, in order to build a formal model of the ACL2 logic, we are faced with deciding whether to take Lisp evaluation or the ACL2 axioms as 'golden' – i.e., as the primary specification.

If the first view were adopted, we could try to build a formal model of ACL2's Lisp evaluation in HOL, so that the ACL2 axioms can be proved consistent with Lisp semantics by, for example, proving `(EQUAL (* (* X Y) Z) (* X (* Y Z)))` always evaluates to `T`. However, the model of Lisp evaluation in HOL would need to be validated against some reference evaluator and it is not clear what this reference should be, since there is no ACL2 definition of S-expression evaluation.

We have decided to adopt the second view, namely that the axioms in the ACL2 source file `axioms.lisp` define the logic [14], rather than some 'golden' evaluator. If there are discrepancies between this and the actual behaviour of ACL2 evaluation (and as far as we know there are none), then our view is that it would be a bug in the evaluator, not in the ACL2 axioms.

Our approach is to define S-expressions in higher-order logic by defining a HOL type `sexp` and then to specify HOL functions operating on this type that correspond to the ACL2 functions axiomatised in `axioms.lisp` (`cons`, `car`, `cdr`, etc.). The key property we must ensure is that for any formula provable in ACL2, its translation is provable in the SEXP theory in HOL. This property guarantees that we can use ACL2 as a trusted oracle for HOL. The property follows from a standard theory interpretation argument: the axioms of SEXP are direct translations of axioms of ACL2, and the rules of inference in HOL are powerful enough to model the first-order and induction rules of inference of ACL2.

4. SEXP: a theory of the ACL2 logic in HOL

We define a HOL theory, called SEXP, which includes a type `sexp` representing S-expressions in higher-order logic and constants corresponding to the ACL2 primitive functions that satisfy the ACL2 logic axioms. The type `sexp` is a recursively defined datatype composed of four kinds of atoms (symbols, strings, characters and complex rational numbers) and pairs of S-expressions. Further details about SEXP can be found in a companion paper [9].

HOL and ACL2 each have their own notions of characters, strings and numbers. Fortunately the match between characters and strings in HOL and ACL2 is exact. In ACL2, numbers are specified axiomatically in `axioms.lisp`, which contain axioms like the associativity and commutativity of addition and multiplication. ACL2 complex rational numbers consist of two rationals: a real part and an imaginary part. Rational numbers in HOL are defined as a quotient type [12] using a rational package developed by Jens Brandt [2] and consist of two integers: the numerator and denominator. Thus an ACL2 number can be represented by four integers (real part numerator, real part denominator, imaginary part numerator, imaginary part denominator). The first-order axiomatisation of numbers in ACL2 admits non-standard interpretations, but the higher-order HOL definition of numbers does not – it constrains numbers to be standard. Thus, there may be properties of numbers in the SEXP theory that cannot be proved in ACL2. We do not view this as a problem as we already know that there are things that can be proved in HOL but not in ACL2.

Not all symbols in the HOL datatype `sexp` correspond to valid ACL2 symbols due to ACL2's rules for 'interning' symbols in packages. This is handled by having an explicit definition of the package structure in HOL and by making the definition of `symbolp` return `nil` on "symbols" which are not symbols according to this structure. Symbols in ACL2 consist of a package name and a symbol name separated by "`::`", but normally only the symbol name is input and output, the package name being implicit via the current package. It simplifies the representation of ACL2 inside HOL if we use fully expanded ACL2 names as the names of the corresponding HOL constants. For convenience when working with HOL we have implemented a mechanism for overloading parser-friendly names onto ACL2 names. For example, `mult`, `add` and `unary_minus` are the parser-friendly HOL names overloaded onto `ACL2::BINARY-*` (multiplication), `ACL2::BINARY-+` (addition) and `ACL2::UNARY--` (unary minus), respectively. These particular names are used in the example in Section 5.

Certain ACL2 symbols represent primitive notions in the initial theory. Examples are `t`, `nil`, `car`, `cdr`, `cons`, `consp` and `if`. There are 33 such primitive symbols:

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

```
t nil acl2-numberp bad-atom<= binary-*
binary-+ unary-- unary-/ < car cdr
char-code characterp code-char complex
complex-rationalp coerce cons consp
denominator equal if imagpart integerp
intern-in-package-of-symbol numerator
pkg-witness rationalp realpart stringp
symbol-name symbol-package-name symbolp
```

The ACL2 logic constrains the interpretation of these by about eighty axioms listed in the file `axioms.lisp`. For example, one of the axioms, called `car-cdr-elim`, is:

```
(defaxiom car-cdr-elim
  (implies
    (consp x)
    (equal (cons (car x) (cdr x)) x)))
```

This uses an auxiliary function `implies` defined in terms of the primitives by:

```
(defun implies (p q) (if p (if q t nil) t))
```

We have defined *all* the ACL2 primitives, and all the auxiliary functions they use, as constants or functions in the HOL theory SEXP. The constants `t` and `nil` are names of particular symbols, and the function `cons` is a primitive constructor of the datatype `sexp`. The functions `car` and `cdr` map `sexp` pairs to their first and second components, respectively, and all other `sexp` values (i.e. strings, characters and numbers) to `nil`. The function `consp` maps pairs to `t` and all other `sexp`-values to `nil`. The function `equal` is defined in HOL by:

```
⊢ ∀x y. equal x y  =  if x = y then t else nil
```

where equality (=) and the conditional are those of the HOL logic. Unfortunately, the HOL parser makes it inconvenient to use "`if`" for the ACL2 conditional on S-expressions in the theory SEXP, because it is a HOL keyword (as illustrated above), so we use "`ite`" (for "if-then-else") instead:

```
⊢ ∀x y z. ite x y z = if x = nil then z else y
```

The auxiliary function `implies` used in the ACL2 axiom `car-cdr-elim` is defined in SEXP by:

```
⊢ ∀p q. implies p q = ite p (ite q t nil) t
```

ACL2 formulae are S-expressions, but HOL formulae are terms of type `bool`. When an ACL2 term `p` is used as a formula it means that `p` is not `nil`, thus we define the HOL formula $\models$ `p` by:

```
⊢ ∀p. ⊨ p  =  ¬(p = nil)
```

to mean that `p` is true. The axiom `car-cdr-elim` is then verified in the HOL model of ACL2 by proving:

```
⊢ ∀x. ⊨ implies
          (consp x)
          (equal (cons (car x) (cdr x)) x)
```

To ensure that our definitions of the ACL2 primitives are sound, we have proved many of the axioms in `axioms.lisp` (we plan to prove them all eventually).

The first-order logic of ACL2 is not typed, but the higher-order logic of HOL4 is. All constants defined in HOL must have a fixed type. For example, the HOL types of the functions in the theory SEXP described above are:

HOL constants	**HOL type**
`t, nil`	`sexp`
$\models$	`sexp→bool`
`car, cdr`	`sexp→sexp`
`cons, equal, implies`	`sexp→sexp→sexp`
`ite`	`sexp→sexp→sexp→sexp`

5. Encoding HOL developments into ACL2

We have implemented a set of tools that can take a sequence of definitions in HOL and create a sequence of definitions in the theory SEXP, together with a proof that the encoding of the definitions is correct. The coding and decoding of HOL functions is performed recursively, using ideas from 'polytypism' [27], by composing the encodings and decodings of sub-functions, starting from a library of encodings and decodings of primitive functions. (e.g. booleans, arithmetic, list-processing, et.).

There is an initial set of functions for encoding/decoding primitive HOL types (e.g. booleans, arithmetic, list-processing etc.) to S-expression. HOL allows new types to be defined recursively, and we have tools that automatically generate encodings to S-expressions from type definitions. For example, consider a HOL type definition:

```
Hol_datatype
  `rose_tree = Branch of (α × rose_tree) list`
```

This defines a new polymorphic unary type operator `rose_tree`, where, for arbitrary type α, the values of type `(`α`)rose_tree` are the empty tree `Branch[]` and non-empty trees `Branch[(`a_1`,`t_1`);...;(`a_n`,`t_n`)]`, where a_i has type α and t_i has type `(`α`)rose_tree` ($1 \leq i \leq n$).

From this HOL type definition, our encoder generates an encoder function for `rose_tree`, parameterised on an encoder for the type parameter α, i.e. a HOL constant:

```
encode_rose_tree:(α →sexp)→(α)rose_tree→sexp
```

If `List:(sexp)list→sexp` encodes HOL lists of S-expressions as ACL2 lists and `nat:num→sexp` is a predefined encoder of HOL natural numbers as S-expressions, then instantiating α to `num`:

```
⊢ encode_rose_tree nat
   (Branch
     [(0,Branch[]); (1,Branch[(2,Branch[])])])
   =
   List[List[nat 0]; List[nat 1; List[nat 2]]]
```

Each HOL type whose values can be encoded as S-expressions also has an automatically generated recogniser, which is a HOL function of type `sexp→sexp` (the S-expression returned will be `t` or `nil`). For example, `consp`, `natp` are recognisers for encoded dotted-pairs and encoded HOL natural numbers, respectively. If a type is parameterised, then its recogniser will be a function that takes recognisers for the type parameters as arguments. For example, the recogniser for S-expressions

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

encoding HOL lists of type `(α)list` is `listp` of type `(sexp→sexp)→(sexp→sexp)`. The recogniser for Cartesian product types $\alpha \times \beta$ is a (curried) function that takes recognisers for α and β as arguments. The definitions of `listp` and `pairp` are given below, using an auxiliary function `andl` that is similar to the ACL2 macro `and` (its definition uses HOL's infix list-cons operator `::`).

```
(andl [] = t) ∧ (andl [s] = s) ∧
(andl (x::y::l) = ite x (andl (y::l)) nil)

listp p x =
 ite
  (equal x nil)
  t
  (andl[consp x; p(car x); listp p (cdr x)])

pairp f g x =
 andl[consp x; f(car x); f(cdr x)]
```

Whenever an encoder for a user-defined HOL type is generated a recogniser is also created.

```
rose_treep f s =
 listp (pairp f (rose_treep f)) s
```

Our encoding tools handle polymorphic type operators, but only ground instances of polymorphic functions can be represented as functions on `sexp` and exported to ACL2. A 'flattening' process generates first-order functions in HOL suitable for exporting to ACL2.

```
nat_rose_treep s =
 ite (equal s nil) t
     (andl
       [consp s; consp (car s);
        natp (car (car s));
        nat_rose_treep (cdr (car s));
        nat_rose_treep (cdr s)])
```

Functions defined in HOL are automatically translated to first-order functions on S-expressions. For example, the counting function `count` defined by:

```
(count (Branch[]) = 0) ∧
(count (Branch ((x,hd)::tl)) =
  1 + count hd + count(Branch tl))
```

is automatically translated to the S-expression function:

```
acl2_count tl =
 ite (nat_rose_treep tl)
     (ite (consp tl)
          (add
            (add
              (nat 1)
              (acl2_count (cdr (car tl))))
            (acl2_count (cdr tl)))
          (int 0))
     (int 0)
```

and a correctness theorem automatically proved:

```
⊢ ∀tl. count tl =
        sexp_to_nat
         (acl2_count(encode_rose_tree nat tl))
```

If a HOL function operates only on datatypes that are in the initial library of encodings to S-expressions, then no type encoding functions need to be generated. For example, the infixed exponentiation operator `**` is defined recursively in HOL by a conjunction of equations, one for the basis and one for the inductive step. Built-in natural number multiplication (`*`) and successor (`SUC`) functions are used.

```
(∀m. m ** 0 = 1)
∧
(∀m n. m ** SUC n = m * (m ** n))
```

The encoder generates a definition of a constant `acl2_exp` that represents `**` as a first-order operation on S-expressions in HOL.

```
acl2_exp m n =
 ite
  (andl [natp n; natp m])
  (ite
    (equal n (int 0))
    (nat 1)
    (mult m
      (acl2_exp m
        (nfix (add n (unary_minus (nat 1)))))))
  (int 0)
```

together with the correctness-validating theorem:

```
⊢ ∀m n. m ** n =
        sexp_to_nat(acl2_exp (nat m) (nat n))
```

Note that the translator generated a definition of the function `acl2_exp` that returns the S-expression corresponding to zero (`int 0`) when either of its arguments are not numbers. Similarly, the predefined SEXP encodings of addition and multiplication return `int 0` on non-number arguments. This ensures that axioms like `ASSOCIATIVITY-OF-*` are true in the HOL model.

6. Moving between ACL2 and HOL

The link between the HOL4 and ACL2 systems is implemented by printing and reading files of S-expressions. To send an S-expression from HOL4 to ACL2 an ML function recursively prints the S-expression to a file suitable for input by ACL2. The reverse direction is similar: ACL2 recursively writes an S-expression to a file that can be input to ML. Before S-expressions are moved between systems they are elaborated into a simple form. For example, HOL terms are simplified by rewriting away auxiliary function like `andl` and `let`-terms are eliminated by β-reductions. ACL2 definitions and formulae have all macros expanded and `lets` replaced by equivalent ACL2 `lambdas`. This elaboration is done securely (i.e. the elaborated forms are provably equivalent to the unelaborated sources).

There are a few low-level details that need care, including the treatment of character, string and number literals. Modelling of ACL2 packages is quite complicated, reflecting the inherent complexity of ACL2 namespace management: see the companion paper [9].

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

The conversion of HOL definitions to S-expressions is straightforward. An equation $f\ x_1\ \cdots\ x_n\ =\ e$ is converted to `(defun` f `(`$x_1\ \cdots\ x_n$`)` $\hat{e}$`)`, where $\hat{e}$ is the translation of e. A HOL theorem $\vdash\ (\models e)$ corresponds to `(defthm` *name* $\hat{e}$`)`, where *name* is generated from the name of the theorem in HOL.

The transfer of S-expressions between the HOL4 and ACL2 systems is not validated by any proof. It is thus vital that the translation code be trustworthy. We have attempted to achieve this by keeping it straightforward. To test our code, we have imported all the ACL2 axioms, definitions and theorems from `axioms.lisp` and also a complete ACL2 model of Y86 (a simple X86-like machine [3]) into HOL, converted them to HOL theorems and definitions, then exported them back to ACL2, and successfully checked that the results are correct. This substantially adds to our confidence that our code for printing and reading S-expressions is sound. Note that this "round trip" test just helps validate the conversions between ML and Lisp, it doesn't prove that the axioms follow by proof in HOL in the SEXP theory (doing this is work in progress).

Definitions and theorems imported from ACL2 can either be trusted, or proof obligations can be established to validate them in HOL. Theorems created in the SEXP theory by trusting ACL2 are 'tagged' with their source, so one can always distinguish theorems proved entirely inside HOL from those proved using ACL2 as an oracle. ACL2 will only admit a definition if it is proved that it is consistent (indeed, conservative) to add it [15]. HOL has a similar discipline. We believe that any definition admitted by ACL2 could also be soundly admitted in HOL (because HOL provides induction support at least as strong as the ϵ_0-induction used by ACL2).

7. Performance

In this section we briefly describe a simple example that shows the kind of performance that can be attained by using ACL2 to execute HOL specifications.

Consider a simple memory model that can interpret 'read' and 'write' instructions of the form `(`b`,`x`,`y`)` where b is a Boolean (`T` for a write, `F` for a read), x is an address (a natural number) and y a value to write (ignored by reads).

The state of a memory is represented in HOL by a list `[(a1,v1);...;(an,vn)]` where `a1`, ..., `an` are the addresses holding non-zero values, and the value stored at $\mathtt{a}_i$ is $\mathtt{v}_i$ ($\mathtt{1} \leq i \leq \mathtt{n}$) – thus an empty list `[]` represents a memory in which all addresses hold `0`.

The function `read_step` takes an address and a memory state and returns the value stored at the address. The function `write_step` takes an address, a value and a memory state and returns an updated memory state with the value written to the address.

A tail-recursive function `run` takes a list of instructions, a memory state and an accumulator consisting of a list of previously read values (in reverse order). It then executes the instructions, adding any values read to the front of the accumulator. A function `make_instrs` generates text programs consisting of sequences of reads and writes starting from a given initial state.

As a benchmark we created a program with a million instructions. The results shown below should be taken as illustrative only, as they involve three different systems (Moscow ML, MLton [33], and ACL2 built on Gnu Common Lisp), and it is not clear if the same things are being timed: the ML compilers report "`gc`", "`sys`" and "`usr`" times (figures given below are `gc+sys+usr` times), whilst ACL2/LISP reports "`real`", "`run-gbc`" and "`gbc`" times (figures given below are `real` times). Timings were done on the same machine.

HOL EVAL	Moscow ML	MLton	ACL2	Common Lisp	ACL2 + stobj
∞?	5.8	1.26	8.16	5.6	0.06

Evaluation by proof in HOL (EVAL) runs out of memory after 10 hours. Translating to ML and using the standard Moscow ML interpreter is a little faster than using the ACL2 read-eval-print loop on the compiled functions and about the same as running them in Common Lisp[1]. The fastest execution is with a manually verified translation to single-threaded code [1]. The MLton ML compiler is the most optimising ML compiler we know, but it is more than an order of magnitude slower than single-threaded ACL2. Execution in ACL2 is within a theorem prover and so the results are based on formal definitions in the logic, unlike those from the ML compilers.

8. Future work and conclusions

We plan to prove all the axioms of the ACL2 logic in HOL. This will verify that we have a sound model of ACL2.

So far we have concentrated on using ACL2 to execute HOL models. The biggest example we have done is a hand translation of the floating point evaluation function that forms part of an ARM co-processor [25]. This executed about 300 times faster in interpreted ACL2 than with HOL's native execution facility (EVAL). Automatic encoding of this is in progress. We plan to use ACL2 to run an existing HOL model of an ARM processor [7]. This will be linked to the floating point co-processor. Using ACL2, it is hoped that we can validate the HOL ARM model against ARM's internal simulator and against actual ARM hardware.

[1] The "Common Lisp" time comes from inserting hand-generated *guards* into the ACL2 functions and evaluating with those functions in the ACL2 loop; but those functions immediately call their compiled Common Lisp counterparts.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

We want to investigate translating HOL goals, and the definitions they depend on, into SEXP. Once the problem has been reformulated into SEXP, we export it to a file of ACL2 S-expressions as an ACL2 *book* that may be processed by ACL2. An ACL2 user then creates an ACL2 book that includes the contents of the one that was generated and perhaps additional intermediate definitions and lemmas. After ACL2 has been led to validate the extended book, it will be trivial to *certify* the generated book, at which point the original SEXP goal can be declared to have been solved and an ACL2-tagged SEXP theorem achieving the goal is created. For this to be sound, we need to be sure that the original goal is a theorem of HOL, but we are confident this is true because the theorems about S-expressions provable in the HOL theory SEXP are a superset of the theorems provable in ACL2. Note that in ACL2 there is an implicit assumption that all instances of induction up to the ordinal ε_0 are available. The HOL logic is sufficiently strong that these instances are provable in SEXP.

Our current treatment of macros may not work for importing a large ACL2 development, say the JVM, into SEXP because elimination of macros can cause significant expansion in code size and loss of mnemonic abstractions. By defining appropriate auxiliary HOL definitions or ML functions, like the functions `List` and `andl` mentioned in Section 5, it may be possible to provide better support for importing macros, but for current work, expanding them seems adequate. Macros require more thought and experimental studies to identify technical issues.

No simplifying assumptions about the ACL2 logic have been made. We believe that if you trust ACL2 and HOL4 then you should be able to trust shared developments using the systems linked according to our approach.

9. Acknowledgements

We have had discussions with many people about this work, including Bob Boyer, Joe Hurd, John Matthews, Pete Manolios, J Moore and Mark Shields. Konrad Slind made major contributions to our ideas and helped us implement tools for defining functions in the SEXP theory.

Warren Hunt and Matt Kaufmann are supported by DARPA and the National Science Foundation under Grant No. CNS-0429591. James Reynolds is supported by an EPSRC Studentship.

References

[1] Robert Stephen Boyer and J Strother Moore. Single-threaded objects in ACL2. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *PADL 2002*, volume 2257 of *Lecture Notes in Computer Science*, pages 9–27. Springer-Verlag, 2002.

[2] Jens Brandt. HOL module `rational`. Website of Reactive Systems Group of the Department of Computer Science at the University of Kaiserslautern. `http://rsg.informatik.uni-kl.de/tools/hol-modules/rational/`.

[3] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[4] Cryptol. Galois Connections, Inc., `http://www.cryptol.net/`.

[5] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In *Fourteenth International Conference on Automated Deduction*, pages 351–365. Springer-Verlag Lecture Notes in Computer Science, 1997.

[6] George Fink, Myla Archer, and Lie Yang. PM: A proof manager for HOL and other provers. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 286–304. IEEE Computer Society, 1991.

[7] Anthony C. J. Fox. Formal specification and verification of ARM6. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 2003.

[8] Ulrich Furbach and Natarajan Shankar, editors. *Proceedings of the third International Joint Conference on Automated Reasoning (IJCAR), Seattle, WA, USA, August 17-20*, volume 4130 of *Lecture Notes in Computer Science*. Springer-Verlag, October 15 2006.

[9] Michael J.C. Gordon, Jr. Warren A. Hunt, Matt Kaufmann, and James Reynolds. An embedding of the ACL2 logic in HOL. In Manolios and Wilding [17].

[10] D. Hardin, E. Smith, and W. Young. A robust machine code proof framework for highly secure applications. In Manolios and Wilding [17].

[11] John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[12] Peter V. Homeier. A design structure for higher order quotients. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2005.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

[13] Joe Hurd. Verification of the Miller-Rabin probabilistic primality test. *Journal of Logic and Algebraic Programming*, 50(1–2):3–21, May–August 2003. Special issue on Probabilistic Techniques for the Design and Analysis of Systems.

[14] Matt Kaufmann and J Strother Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997. See URL `http://www.cs.utexas.edu/users/moore/publications/km97a.pdf`.

[15] Matt Kaufmann and J Strother Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[16] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[17] Panagiotis Manolios and Matthew Wilding, editors. *Proceedings of the Sixth International Workshop on the ACL2 Theorem Prover and its Applications*, Seattle, Washington, August 2006. ACM Digital Library.

[18] Ian A. Mason and Carolyn L. Talcot. IOP: The InterOperability Platform & IMaude: An interactive extension of Maude. In *5th International Workshop on Rewriting Logic and its Applications*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 315–333. ScienceDirect, 2005.

[19] S. McLaughlin. An interpretation of Isabelle/HOL in HOL Light. In Furbach and Shankar [8].

[20] Tobias Nipkow, L. C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, London, UK, 2002.

[21] Michael Norrish and Konrad Slind (project administrators). The HOL4 System. SourceForge website. `http://hol.sourceforge.net/`.

[22] Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Furbach and Shankar [8].

[23] L. Pike, M. Shields, and J. Matthews. A verifying core for a cryptographic language compiler. In Manolios and Wilding [17].

[24] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, jun 1995. Springer-Verlag.

[25] James Reynolds. A HOL implementation of the ARM FP coprocessor programmer's model. Technical Report number PRG-RR-05-02 (TPHOLs2005 Emerging Trends Proceedings), Oxford University Computing Laboratory, 2005. `http://web.comlab.ox.ac.uk/oucl/conferences/TPHOLs2005/emerging-trends.pdf`.

[26] Rockwell Collins, Inc. press release. CEDAR RAPIDS, Iowa, August 24, 2005. Rockwell Collins receives MILS certification from NSA on microprocessor. `http://www.rockwellcollins.com/news/page6237.html`.

[27] Konrad Slind and Joe Hurd. Applications of polytypism in theorem proving. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 103–119, Rome, Italy, September 2003. Springer.

[28] Mark Staples. Linking ACL2 and HOL. Technical Report UCAM-CL-TR-476, University of Cambridge Computer Laboratory, November 1999. `http://www.cl.cam.ac.uk/users/mjcg/hol2acl2/papers/staples99linking.ps`.

[29] Kong Woei Susanto. Verification platform for system on chip. Web page hosted at Glasgow University. `http://www.dcs.gla.ac.uk/~susanto/vp.htm`.

[30] Kong Woei Susanto and Tom Melham. An AMBA-ARM7 formal verification platform. In Jin Song Dong and Jim Woodcock, editors, *Formal Methods and Software Engineering: 5th International Conference on Formal Engineering Methods, ICFEM 2003: Singapore, November 5–7, 2003: Proceedings*, volume 2885 of *Lecture Notes in Computer Science*, pages 48–67. Springer-Verlag, 2003.

[31] Thomas C. Hales. The Flyspeck Project Fact Sheet `http://www.math.pitt.edu/~thales/flyspeck`.

[32] Thomas Tuerk and Klaus Schneider and Mike Gordon. Model Checking PSL using HOL and SMV. Submitted for publication June 2006.

[33] Stephen Weeks. The MLton whole-program, optimizing Standard ML compiler. `http://mlton.org/`.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

ACL2SIX : A Hint used to Integrate a Theorem Prover and an Automated Verification Tool

Jun Sawada
IBM Austin Research Laboratory
Email: sawada@us.ibm.com

Erik Reeber
University of Texas at Austin
Email: reeber@cs.utexas.edu

Abstract— **We present a hardware verification environment that integrates the ACL2 theorem prover and SixthSense, an IBM internal formal verification tool. In this environment, SixthSense is invoked through an ACL2 function `acl2six` that makes use of a general-purpose external interface added to the ACL2 theorem prover. This interface allows decision procedures and model-checkers to be connected to ACL2 by simply writing ACL2 functions. Our environment also exploits a unique approach to connect the logic of a general-purpose theorem prover with machine designs in VHDL without a language embedding. With an example of a pipelined multiplier, we show how our environment can be used to divide a large verification problem into a number of simpler problems, which can be verified using automated verification engines.**

I. Introduction

Formal verification technology can be grouped into two major categories, automated verification techniques and interactive theorem proving techniques. Automated verification techniques can verify a system without much human involvement, but fail to scale to large systems. On the other hand, interactive theorem proving techniques can scale, but require significant human efforts.

The combination of these two techniques can create an ideal tool for formal verification. We can use automated verification techniques to verify properties of machine components, and then use theorem proving to combine the results. Or we can verify the correctness of algorithms using theorem proving, and then use automated verification techniques to check that the algorithms are correctly implemented in the hardware.

In order to make the verification tool practical, it is essential to access a hardware description language (HDL). It is preferable to access an HDL that is already used by industrial designers, such as Verilog or VHDL. For this reason subsets of Verilog and VHDL have been embedded into the ACL2 theorem prover on a number of occasions [5], [13], [14]. Embedding even a core subset of such an HDL into the formal language of a general-purpose theorem prover, however, has proven to be tedious and difficult.

Another problem with embedding an HDL into a theorem prover is that the theorem prover ends up with a very low-level model of the hardware. We think this is not the best place to apply theorem proving techniques. Rather theorem proving is best applied at a more abstract level, such as during the verification of the fundamental algorithms that perform mathematical operations.

Our strategy uses an automated tool to verify properties about low-level circuits. The use of the theorem prover is directed toward high-level work such as combining those results and verifying algorithms. That way, we lessen the amount of theorem proving effort, and enable the automatic reverification of hardware after making any low-level modifications. Furthermore we have, for the most part, avoided embedding the formal semantics of an HDL in a general-purpose theorem prover.

In this paper, we focus on the approach and techniques used in combining the ACL2 theorem prover [8], [6] and SixthSense, an IBM internal verification tool [10]. In this combination, we translate ACL2 properties to be checked into VHDL, instead of embedding the VHDL hardware model into the ACL2 logic. SixthSense then compiles the VHDL for the property as well as the hardware under test, and runs verification algorithms to check the property on the hardware. In a sense, SixthSense is used as ACL2's interface to VHDL, hiding all the details of the low-level design.

This combined tool is built on top of a small extension of the ACL2 theorem prover, which allows connection to external tools. This extension itself is general purpose, and not specific to SixthSense. Thus other tools can be connected to the ACL2 theorem prover by simply providing an ACL2 function similar to the one described in this paper, without further changing the source code of the ACL2 theorem prover.

After introducing minimal facts about ACL2 and SixthSense in Section II, we discuss how we build the connection of those tools in Section III and Section IV. In Section V, we explain the verification of a pipelined multiplier using this combined tool. We cover related work in Section VI and conclude in Section VII.

II. Background

This section provides a quick overview of the ACL2 theorem prover and the SixthSense verification tool.

The ACL2 logic, the language of the ACL2 theorem prover, is an applicative subset of Common Lisp with certain extensions. All functions and theorems are written in typical Lisp-like prefix notation. We assume some knowledge of Common Lisp in this paper. A basic ACL2 tutorial can be found on the web [7].

In the ACL2 logic, all functions are defined with `defun`, as in Common Lisp. ACL2 can be used to create quite complex

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

programs. In fact, the majority of the ACL2 system itself is written in the ACL2 logic. This is made possible by a program mode in which functions may be defined without extending the ACL2 theory. These functions have no logical meaning for the theorem prover and no properties can be proven about them. The system-call function, for example, is a program-mode function.

Theorems are defined using the `defthm` event. Often a `defthm` takes the form of

(defthm *thm-name thm-body* :hints *hint-list*).

This `defthm` tries to prove the expression *thm-body* with ACL2 proof algorithms, and stores it with *thm-name* if it is successfully proven. During the course of the proof, ACL2 might be guided by hints in *hint-list*. Each hint is of the form (*goal-spec* :key_1 val_1 ... :key_n val_n), where *goal-spec* specifies the node in the proof tree to apply the hint, and the key-value pairs specify what actions to be taken. For example, the hint `("Goal" :use theorem1)` directs ACL2 to apply `theorem1` at the beginning of the proof, as `"Goal"` refers to the root of the proof tree.

Although an extensive programing capability is provided in the system, ACL2 does not allow the user to extend the theorem proving algorithms. In this way it differs from HOL [2], which uses user-defined functions to implement theorem proving algorithms. We will explain how we change this limitation in the following section.

SixthSense is a powerful IBM internal industrial verification tool. The target hardware and the checked properties are provided as finite state machines written in VHDL or Verilog. SixthSense can prove safety properties that always hold, or present counter-examples, as waveforms, to safety properties that do not. SixthSense can in effect prove many types of properties by reformulating them into safety properties, including bounded LTL model-checking properties and properties specifying that a condition holds after a specific number of clock cycles.

SixthSense employs a transformation-based verification approach[10]. It has numerous automated verification engines that take a verification problem and either prove it, or convert it to a simpler problem and pass it to the next engine. Some of the engines used in SixthSense are:

- Redundancy removal engine: Identifying functionally redundant gates and removing them.
- Re-timing engine: Moving logic gates beyond hardware latches, in an attempt to reduce the number of latches.
- Structural target enlargement engine: Using preimage computation to enlarge the target states.
- Localization engine: An over-approximate transformation that isolates a cut point of the net-list and replaces it by primary inputs.
- Semi-formal search engine: Searching for counter examples in symbolic and random simulations.
- SAT engine: Solving the problem by using a SAT solver.

SixthSense applies these engines successively in an attempt to verify a property. If it cannot solve a verification problem with the current engine, it passes a simplified verification problem to the next engine. Some of the engines such as the SAT engine are terminating, and either solve the problem or fail.

By default, SixthSense's expert-system mode intelligently applies various engine sequences. If a certain engine sequence does not look promising, it backtracks to an earlier stage and applies different engines with different parameters. Alternatively, one can provide SixthSense with a configuration file that guides it to use engines in a specified order with specific parameters.

III. `:External` Hint Extension to ACL2

In order to combine ACL2 and SixthSense into one environment, we implemented an interface between the ACL2 theorem prover and outside procedures. This extension is implemented as a new ACL2 hint, which is invoked by the keyword `:external`. Like all other ACL2 hints, `:external` reduces a conjectured ACL2 term to other forms that are supposed to be easier to prove. The `:external` hint does this reduction by calling a user-defined ACL2 function.

A function *fn* called through `:external` hint should have the type signature of (*fn* `args cl state`) => (`mv err lst state`)—i.e. *fn* takes an optional argument `args`, a clause to be proven `cl` and an ACL2 state `state` and returns a triple of error flag `err`, a list of new clauses `lst` and an updated ACL2 state. Note that the ACL2 primitive `mv` creates a multi-valued structure, which is then accessed through `mv-let`—for example, `(mv-let (x y) (mv a b) x)` is equal to `a`.

ACL2 internally represents a clause $l_1 \vee l_2 \vee \cdots \vee l_n$ by a list of literals $(l_1\ l_2\ \ldots\ l_n)$. The function *fn* is supposed to reduce the clause into a set of simpler clauses, and return them as a list `lst`. The reduction should be such that proving all the clauses in `lst` is sufficient to derive the original clause `cl`.

We provide an example of the `:external` hint below.

```
(defun split-a-and-b (cl state)
  (mv nil
      (list
       ;; when a is not bool
       (append '((booleanp a)) cl)
       ;; when b is not bool
       (append '((booleanp b)) cl)
       ;; when a and b
       (subst ''t 'b (subst ''t 'a cl))
       ;; when ~a and b
       (subst ''t 'b (subst ''nil 'a cl))
       ;; when a and ~b
       (subst ''nil 'b (subst ''t 'a cl))
       ;; when ~a and ~b
       (subst ''nil 'b (subst ''nil 'a cl)))
      state))

(defthm theorem-1
 (implies
  (and (booleanp a) (booleanp b))
  (or (not b) (and a b) (and (not a) b)))
 :hints (("Goal" :external (split-a-and-b))))
```

When called through an :external hint, the function split-a-and-b splits cl into six cases depending on the values of the variables a and b. The first two are for the case where a and b are not Boolean, and this case is proven by clauses (booleanp a) ∨ cl and (booleanp b) ∨ cl, respectively. Each of the following four cases substitutes the truth value t and the false value nil for a and b. This hint proves a tautology of a and b such as theorem-1, by simply trying all combinations of truth values. Note that the :external hint passes arguments cl and state to the hint function implicitly.

Although this example is very simple, a user can write a more detailed theorem proving procedure as a new function, and call it through the :external hint interface. This way, the user can extend the capability of the ACL2 theorem prover. Since the function called through :external takes an ACL2 state as an argument, it can also search the ACL2 database for already proven theorems and apply them to the conjectured clause.

Additionally, the external hint can be used to call any decision procedures, model checkers, or other theorem provers. The :external hint function can also write or read to a file system. We use this as the foundation of our interface between ACL2 and SixthSense.

On the other hand, the :external hint can be a very dangerous feature. Since any function can be called from the :external hint, a user may end up proving an invalid theorem. For example, a hint function:

```
(defun wrong-hint (cl state)
  (mv nil nil state))
```

reduces the given clause cl to an empty list of clauses, and "proves" any theorem. A user must be extremely cautious in using the :external hint capability.

Although our :external hint interface does not provide an LCF-style type-system to guarantee the correctness of user defined extensions, we can mitigate the dangers of the :external hint by adding tags to theorems. Similar tags are used in the HOL theorem prover [3] for the connection to external decision procedures. Such tags allow the user to control and check which :external hint extensions have been used for the verification of their theorems.

IV. Interface to SixthSense

We defined an ACL2 program-mode function acl2six, which is called by this :external hint interface. This extension is used to verify that a property, written in the ACL2 logic, holds for a hardware design described in VHDL. The acl2six function translates an ACL2 property that we want to verify to VHDL, and passes it to the SixthSense verification tool, which in turn attempts to verify that the VHDL design satisfies the property. If the property is verified by SixthSense, then ACL2 uses this result to further its proof effort. If SixthSense fails to verify the property, then one can use a waveform viewer to examine the counter-example generated by SixthSense.

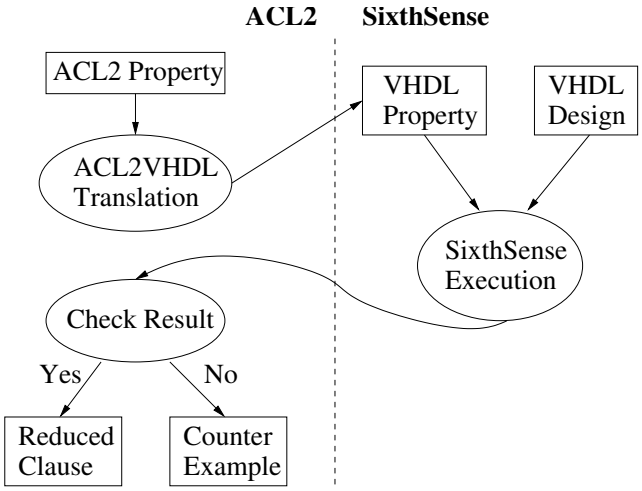


Fig. 1. The flow of acl2six function call. A rectangle represents data, whereas an ellipse represents a process.

To describe the implementation of acl2six in detail, we first introduce two new function symbols sigbit and sigvec, with the following type signatures:

```
(sigbit entity signame cycle phase)
=> bit

(sigvec entity signame lbit hbit cycle phase)
=> bv
```

These functions output a single-bit value bit or bit-vector value bv of a signal in our hardware design, given a machine model entity, the signal name signame, integer pair lbit and hbit providing the lowest and highest index of a bit-vector, and natural numbers cycle and phase denoting the time.

The machine model entity not only designates the machine module called "entity" in VHDL, but also specifies its interface, its initial state, the primary inputs to the machine, and its clocking information. Essentially, entity denotes the machine and its execution environment. Functions sigbit and sigvec return the value of a signal in that particular run at the given time. Time is designated by the pair of cycle and phase, where cycle increments from 0, and each clock cycle is divided into two phases denoted by 1 and 2.

A data flow diagram of the acl2six function is shown in Fig. 1, which proceeds through the following steps:

- **Translate property to VHDL**. The acl2six function uses simple heuristics to pick the ACL2 term that will be checked by SixthSense. This term is then translated into a VHDL function that inputs the given sigvec signals and produces a one bit signal that is high when the term holds. The user may also manually select a term for translation. The ACL2VHDL translator [15] is used to convert the ACL2 term to VHDL. Properties must be written in the subset of the ACL2 language supported by the ACL2VHDL translator. This subset includes a library of basic logical primitives, including bit negation (b-not), bit conjunction (b-and), bit-vectors negation (bv-not),

bit-vector conjunction (`bv-and`), addition (`bv+`), shift, sub-vector extraction, and many others. For each primitive, we have a definition in ACL2 and a corresponding definition in VHDL. The translator merely substitutes each ACL2 function with its corresponding VHDL function.

- **SixthSense Execution**. After creating a configuration and driver file to control the SixthSense run, `acl2six` executes SixthSense by using ACL2's system call function. SixthSense then checks that the given property is always true.
- **Interpreting SixthSense Results**. If SixthSense successfully verifies its property, a new clause list is returned to the theorem prover. If the original clause is $(l_1\ l_2\ \ldots\ l_n)$ and property l' is proven by SixthSense, `acl2six` returns $(\neg l'\ l_1\ l_2\ \ldots\ l_n)$ as the new clause. In other words, the theorem prover now has to prove the original formula assuming the property proved by SixthSense.

  If SixthSense fails to verify the property, a counter-example to the checked property is usually provided. The user can analyze it by using a GUI-based waveform-viewing tool. In order to help the user debug it, the value of arbitrary ACL2 expressions can be also viewed in the wave-form, if those expressions are provided to the `acl2six` hint.

Note that the entire `acl2six` process can be written as an ACL2 function.[1] Except the implementation of the general-purpose `:external` hint feature, we did not modify the underlying ACL2 code. This suggests that we can use the same `:external` mechanism to connect other decision procedures by simply writing new interface functions. Considering the complexity of the underlying ACL2 source code, this makes it much easier to write a similar extension. The extension function itself should be carefully written to avoid semantic errors, but at least we do not have to endanger the soundness of the core algorithm of ACL2.

Part of the uniqueness of our approach is in its use of a translator from ACL2 to VHDL, rather than the other way around. This allows us to write a relatively simple translator from ACL2 functions to VHDL functions, rather than a more complex translator from VHDL designs to equivalent ACL2 models. Note that this approach does not put restrictions on the designs being verified.

Another merit of our approach is that the theorem prover does not have to deal directly with the low-level details of hardware designs. These details change frequently, putting a heavy maintenance burden on user-guided proofs concerning them. In our approach theorem provers can focus on what they do best, leaving automated tools to check the design changes.

[1] At the moment of writing this article, ACL2VHDL translation written in normal LISP is called as a separate process, but it can be rewritten as ACL2 functions.

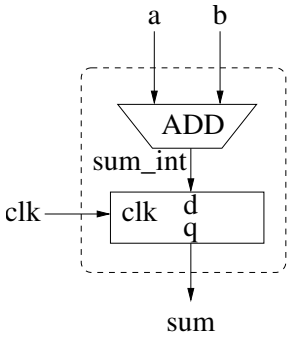


Fig. 2. A 32-bit adder model

```
(defun add32 ()
    '(add32
      (port
       (clk :in std_ulogic)
       (a :in std_ulogic_vector (0 31))
       (b :in std_ulogic_vector (0 31))
       (sum :out std_ulogic_vector (0 31)))
      (extra-assigns  (clk "c0"))))

(defthm adder-adds
  (implies
   (and (integerp n) (<= 1 n))
   (equal
    (bv+ (sigvec (add32) a (0 31) (1- n) 2)
         (sigvec (add32) b (0 31) (1- n) 2))
    (sigvec (add32) sum (0 31) n 2)))
  :hints
  (("Goal"
    :external
    (acl2six ((:cycle-expr n)
              (:ignore-init-cycles 1))))))
```

Fig. 3. The entity definition of the adder and its proof script.

A. Adder Example

We use the simple adder in Fig. 2 as an example. This adder takes two 32-bit inputs, `a` and `b`, and produces their 32-bit, `sum`, when `clk` is triggered.

The verification script for this adder is shown in Fig. 3. The function `add32` defines the `entity` information as a list of the VHDL entity name, its interface, and its clocking information. The input `clk` is driven by a built-in clock source called `c0`. The function `add32` definition is "disabled" to prevent the theorem prover from expanding it. This entity definition allows us to succinctly write the `sigvec` signal expression, even for a hardware module with many interface ports and many other execution conditions. Although this might seem a minor point, it is important to be able to write all of the detailed execution environment in a short form for an industrial setting. Often such details can affect the validity of the tested properties, and it is desirable that the user not have to repeatedly specify them.

The `defthm` expression in Fig. 3 shows a theorem that can be proven using our system. The `sigvec` expression

```
(sigvec (add32) sum (0 31) n 2)
```

denotes the value of the 32-bit vector signal `sum` at cycle n in phase 2. The `sigvec` expressions involving `a` and `b` are the same, except that their values are observed at cycle $n-1$. The theorem `adder-adds` proves that the output `sum` is equal to the modular summation of the inputs `a` and `b` from the previous cycle. Typical properties checked by our system are like the conditions involving signals at different timing. In this example, we use only the input and output signals, but we can also prove theorems about internal signals, such as `sum_int` in Fig. 2.

When the `defthm adder-adds` is executed, the `acl2six` function processes the theorem following steps:

- **Translate property to VHDL**. The following term is selected for translation:

```
(equal
 (bv+ (sigvec (add32) a (0 31) (1- n) 2)
      (sigvec (add32) b (0 31) (1- n) 2))
 (sigvec (add32) sum (0 31) n 2))
```

  The function `bv+` is a logical primitive, with a VHDL definition known to ACL2VHDL and the `sigvec` functions are interpreted as references to the adder hardware design.
  Note that `add32` does not provide the initial state, or explain how to drive the input ports `a` and `b`. Unless otherwise specified, `acl2six` considers all possible combination of initial state and input values. Internally by SixthSense, they are assigned new input variables.
- **SixthSense Execution**. SixthSense is now asked to verify the created VHDL property. Note that since the first `sum` comes out a cycle after the machine starts, the property does not hold in cycle zero. The argument `(:ignore-cycles 1)` tells SixthSense to ignore the first cycle.
- **Interpreting SixthSense Results**. SixthSense returns that the property holds for all time after the first cycle. The `acl2six` function then adds a new hypothesis to represent this statement. The theorem prover then finishes the proof by showing that this new hypothesis implies the original theorem.
  Note that if a larger `:ignore-cycle` argument were provided, however, the property verified by SixthSense would be too weak to imply `adder-adds` and the proof attempt would fail.

B. Soundness

In this sub-section we present a proof sketch showing that the use of an `acl2six` hint does not result in an inconsistent theory. Our strategy is to show that any theorem admitted by the `acl2six` hint can be theoretically proven by the ACL2 theorem prover.

Since the hardware design is a finite state machine, we can express its behavior in ACL2 as a recursive function, $stepn(s, i, n)$, where s is an initial state, i is an input sequence, and n a cycle number. The definitions of `sigbit` and `sigvec` then return a value of signals based on this function. Let us also define an ACL2 function $M(S, I, n)$ that returns the set of reachable states of the machine by enumerating $stepn(s, i, n)$ for all possible $s \in S$ and $i \in I$ at cycles up to n.

Now let us show that any theorem t_i, proven by `acl2six` is also provable by the pure ACL2 theorem prover. Since every property that can be proven by SixthSense is a safety property, t_i can be reformulated using a predicate P to $x \in M(S, I, n) \rightarrow P(x)$ for arbitrary cycle n. Because this is a finite state machine, $M(S, I, n_0) = M(S, I, n)$ for all n larger than some n_0, which can be proven by ACL2 using induction. Since SixthSense proves the VHDL equivalent of $P(x)$ holds for all reachable states, ACL2 can prove that $x \in M(S, I, n_0) \rightarrow P(x)$ by evaluating $P(x)$ for every $x \in M(S, I, n_0)$. Combining these results, ACL2 can prove that t_i holds. QED.

In this argument, we make a few assumptions:

- SixthSense does not prove any properties that are not really valid in the hardware design.
- All functions defined in both VHDL and ACL2 (such as `bv+`), have equivalent semantics.
- The constrained functions `sigbit` and `sigvec` are never instantiated. ACL2 allows a constrained function f_c to be instantiated with another function f if f satisfies all the constraints of f_c. Although we implement `sigbit` and `sigvec` as constrained functions, they have a specific meaning with respect to the hardware design, and should therefore never be instantiated with another function.

V. Multiplier Example

We have used our integration of ACL2 and SixthSense to verify a pipelined 54x53 bit multiplier design, which is used in an industrial floating point unit. The verification of this multiplier is described in detail in a previous paper[12]. Here we describe the overall approach used.

An overview of the multiplier design is shown in Fig. 4. The inputs **A** and **C** first go through the Booth encoding stage, which produces 27 partial sum vectors that when summed are equal to the multiplication of **A** and **C**. These 27 vectors are then compressed during successive stages into 18, 12, 6, 4, and finally 2 bit-vectors, **Sum** and **Carry**, which when summed are equal to the multiplication of **A** and **C**. The final summation is not implemented in the multiplier itself, as it is a typical design in a floating-point unit. The multiplier requires four and a half clock cycles, starting at the beginning of a cycle and ending by the middle of one four cycles later.

The correctness of this design is encoded by the following ACL2 theorem:

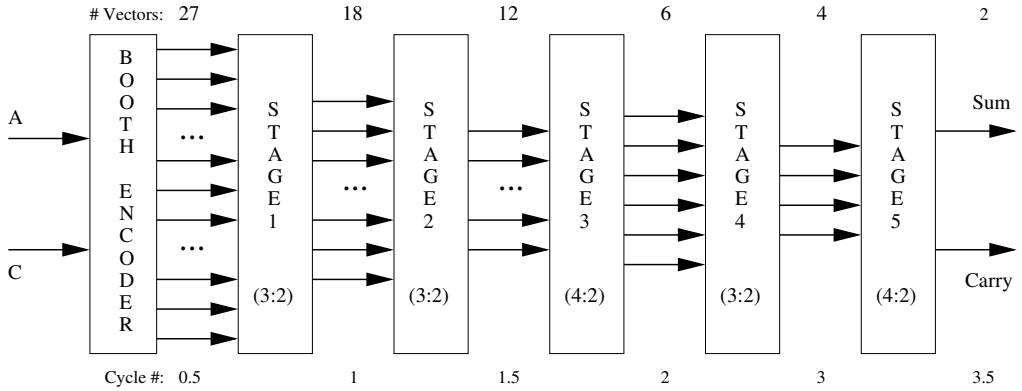


Fig. 4. An overview of the multiplier design.

```
(defthm multiplier-correct
 (implies
  (and (integerp n)
       ; extra 2 cycles needed
       ; for initialization
       (<= 7 n))
  (equal
   (bv+ (Sum-output n 1)
        (Carry-output n 1))
   (bv (* (bv-val (A-input (- n 4) 2))
          (bv-val (C-input (- n 4) 2)))
       108))))
```

Note that the function `bv-val` returns the natural number that a bit vector represents, and `bv` converts a natural number into its corresponding bit-vector. Also, the function `bv+` returns a bit vector representing the summation of its two input bit vectors. The macros `A-input`, `C-input`, `Sum-output`, and `Carry-output` produce the corresponding input and output signals of the multiplier at a given cycle and phase. These macros are defined by using `sigbit` and `sigvec`.

Multiplier designs are particularly difficult to verify with SixthSense, since it does not implement any algorithms specialized for multipliers. This makes the multiplier a good example to illustrate how we can verify designs using the combination of ACL2 and SixthSense.

Our proof begins by verifying the algorithm used in the Booth encoding multiplier. The definition of a Booth encoding multiplier and some proof scripts are shown in Fig. 5. The encoder computes the multiplication of two n bit vectors x and y by summing $n/2$ partial sums, each of which consist of either $2 * y$, y, 0, $-y$, or $-2 * y$ shifted to the left. Function `acl2-Booth-vector` computes a single partial sum, and `acl2-Booth-mult` adds all of them. The correctness of the multiplier is given as a theorem `simple-Booth-encoder-correct`. The proof is conducted by first proving the intermediate lemma `simple-Booth-encoder-lemma` by mathematical induction, and then substituting `(* 2 x)` for `x`. This arithmetic proof on the Booth algorithm is a good example where the use of the ACL2 theorem prover is most suitable.

The next step in the verification of the multiplier is to check the partial sums generated by the real Booth encoder stage is

```
;; We're really considering the two's
;; complement encoding of (mod x 8).
(defun acl2-Booth-vector (x y)
  (case (mod x 8)
    (3 (* 2 y))
    (2 y)
    (1 y)
    (0 0)
    (7 0)
    (6 (- y))
    (5 (- y))
    (4 (* -2 y))))

;; Each recursive step involves a two bit
;; right-shift of x and a two bit
;; left-shift of y.
(defun acl2-Booth-mult1 (x y)
  (if (zp x)
      0
    (+ (acl2-Booth-vector x y)
       (acl2-Booth-mult1 (floor x 4)
                         (* 4 y)))))

;; The multiplication begins by adding
;; a zero to x.
(defun acl2-Booth-mult (x y)
  (acl2-Booth-mult1 (* 2 x) y))

(defthm simple-Booth-encoder-lemma
  (implies
   (and (natp x)
        (integerp y))
   (equal (acl2-Booth-mult1 x y)
          (* (floor (1+ x) 2) y))))

(defthm simple-Booth-encoder-correct
  (implies
   (and (natp x)
        (integerp y))
   (equal (acl2-Booth-mult x y)
          (* x y))))
```

Fig. 5. The definition of a simple Booth encoder, along with the main lemma required to verify it in the ACL2 theorem prover, and a theorem stating that it performs multiplication.

```
(defthm Stage1-compressor-8-correct
  (implies
   (and (integerp n)
        (<= 4 n))
   (equal
    (bv+ (stage1-output-sum8 n 2)
         (stage1-output-carry8 n 2))
    (bv+ (vhdl-Booth-output-26 n 1)
         (bv+ (vhdl-Booth-output-25 n 1)
              (vhdl-Booth-output-24 n 1)))))
  :hints (("Goal" :external (acl2six ...))))
```

Fig. 6. `Stage1-compressor-8-correct` uses SixthSense to verify that the summation of the inputs to the 8th stage 1 compressor is equal to the summation of its three outputs

equivalent to the one generated by `acl2-Booth-vector`. This is a little harder than we thought, because the real Booth encoder implements a number of tricks to optimize the design. Also we have to define an intermediate function using the ACL2VHDL library, as the functions in Fig. 5 cannot be directly fed to the `acl2six` hint. By defining such functions and calling SixthSense via `acl2six`, we prove that the real Booth encoder generates the correct partial sums. The theorem prover then composes these results to prove that the summation of the Booth encoder output signals is equal to the multiplication of the original inputs.

To complete the verification of the multiplier, we must prove that the summation of all the vectors are preserved at each compression stage. We first verified that each carry-save adder reduces three or four inputs into two outputs while preserving the sum. `Stage1-compressor-8-correct`, in Figure 6, is an example of such a lemma, which is verified by SixthSense automatically. We next used the ACL2 theorem prover to combine all the lemmas together. It repeatedly applied rewriting of the summation term using commutativity and associativity of `bv+` rules to prove the preservation of the sum. The results from the three steps are combined to prove the theorem `multiplier-correct`.

VI. Related Work

There have been many integrations between model-checking and theorem proving tools. The general-purpose theorem prover PVS was built with model checking as a primitive proof engine [11]. The SyMP model prover uses a more general approach to integrating the two techniques [1]. With the ACL2 theorem prover, numerous model-checking inspired engines have been integrated, most recently UCLID [9]. What makes our system unique is the fact that we are verifying, in a scalable manner, industrial RTL-level designs written in an HDL.

Within the ACL2 theorem proving community there are three systems of which we know that verify RTL-level designs in HDL: AMD's system that translates Verilog to ACL2 [14], Hunt and Reeber's system that translates Verilog to DE2 [5], and Borrione's system that translates VHDL to ACL2 [13]. All of these use the embedding of HDL in ACL2 logic, making our approach unique. And of these only AMD's system captures a broad enough range of the RTL to capture many industrial designs.

Outside the ACL2 theorem proving community, FORTE [17], developed by Intel and descended from HOL-Voss [4], is the most well-known verification system closely related to our approach. It combines a symbolic trajectory evaluator (STE) and an LCF-style higher-order-logic theorem prover called ThmTac to verify a broad range of industrial RTLs. Both FORTE and our `acl2six` systems use similar approaches to the verification problem, using the automated verification engine for fast changing hardware design, while theorem provers are used for verifying high-level specifications that are less likely to change and can be reused.

However, there are a few critical differences between FORTE and our approach. The first difference is the theorem prover. FORTE uses what they repeatedly describe a "light-weight" theorem prover. It is mostly used for stitching individual STE verification results together, and less emphasis is put on the proof of high-level concepts. On the other hand, we combined a general-purpose theorem prover to an industrial verification tool. A full-featured prover is beneficial, especially one takes a progressive approach to verifying hardware properties. The beginning user might only use the prover for case-splitting and result stitching. An advanced user, however, can gradually use more features, eventually proving more difficult and mathematically deep theorems.

Another difference is that FORTE integrates the underlying STE as a white-box to the theorem prover, while we treat our underlying SixthSense system as a black-box. The developers of FORTE claim that their tight integration gives them access to the internal data structures, allowing better debugging and tuning. We do not agree with this, as our black-box approach can also control the SixthSense's parameters through configuration files, and also use the low level debugging tools. Rather we see more advantages for black-boxing. For example, the SixthSense calls can be repeated independently of the theorem prover, allowing us to pass the failed information to the engineers who do not wish to run the ACL2 theorem prover. Furthermore our generic integration approach can, in principle, call other model checkers on demand; by loading another definition of an integration function similar to `acl2six`.

Finally, FORTE incorporates the VHDL as a netlist into the system after compilation. This is similar to other language embedding approach. Our system only incorporates the proven properties into the theorem prover.

VII. Conclusion

The integration of ACL2 and SixthSense is implemented as an ACL2 function `acl2six` on top of a simple new ACL2 hint, `:external`. Only 57 lines of the theorem prover source code were modified to implement this new hint. The implementation of `acl2six`, on the other hand, consists of over 3000 lines of ACL2 functions. These functions form a library, whose compiled code can be dynamically loaded. In this way, we cleanly separate the `acl2six` implementation from the rest of the ACL2 system.

We avoided embedding the VHDL language into the ACL2 logic by using the ACL2VHDL translator, which translates a subset of ACL2 to VHDL, along with a simple machine model based on the `entity` data-structure and the constrained functions `sigbit` and `sigvec`. While our method restricts the properties that can be proven automatically, it allows for a broad range of models—i.e. all VHDL designs currently understood by SixthSense.

In Section V we describe how our system was used to verify a pipelined multiplier, but we believe our system will scale well to much larger designs. The verification of the entire floating-point unit, relative to the IEEE-754 standard would be an interesting application of our system. Considering the richness of the arithmetic analysis required for the divide and square root verification [16], the combination of theorem proving and fully automated verification should shine in this application.

A. Acknowledgements

We would like to acknowledge Sandip Ray, for building the initial prototype of the `acl2six` system; Matt Kaufmann, for helping design the `:external` extension of ACL2; and Jason Baumgartner, Hari Mony, and Viresh Paruthi, for their help with the SixthSense tool.

REFERENCES

[1] S. Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.

[2] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[3] E. L. Gunter. Adding external decision procedures to hol90 securely. In *Theorem Proving in Higher Order Logics*, pages 143–152, 1998.

[4] J. J. Joyce and C.-J. H. Seger. The HOL-Voss System: Model-Checking inside a General-Purpose Theorem-Prover. In *Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 185–198. Springer-Verlag, 1994.

[5] W. A. H. Jr. and E. Reeber. Formalization of the DE2 Language. In *CHARME*, pages 20–34, 2005.

[6] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, 2000.

[7] M. Kaufmann and J. S. Moore. *ACL2: A Computational Logic for Applicative Common Lisp*. URL:http://www.cs.utexas.edu/users/moore/-acl2/acl2-doc.html.

[8] M. Kaufmann and J. S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, apr 1997.

[9] P. Manolios and S. K. Srinivasan. Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements. In *DATE*, pages 168–175, 2004.

[10] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann. Scalable Automated Verification via Expert-System Guided Transformations. In A. J. Hu and A. K. Martin, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2004*, volume 3312 of *LNCS*, pages 217–233. Springer-Verlag, 2000.

[11] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In *CAV*, pages 411–414, 1996.

[12] E. Reeber and J. Sawada. Combining ACL2 and an Automated Verification Tool to Verify a Multiplier. In *Sixth International Workshop on the ACL2 Theorem Prover and its Applications (ACL2 2006)*, 2006.

[13] V. M. Rodrigues, D. Borrione, and P. Georgelin. Using the ACL2 Theorem Prover to Reason about VHDL Components. *RITA*, 7(1):129–148, 2000.

[14] D. Russinoff. A Mechanically Checked Proof of IEEE Compliance of a Register-Transfer-Level Specification of the AMD K7 Floating Point Multiplication, Division and Square Root Instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[15] J. Sawada. ACL2VHDL Translator: A Simple Approach to Fill the Semantic Gap. In *Proceedings of the Fifth International Workshop of the ACL2 Theorem Prover and its Applications (ACL2-2004)*, 2004.

[16] J. Sawada and R. Gamboa. Mechanical Verification of a Square Root Algorithm Using Taylor's Theorem. In *Formal Methods in Computer Aided Design (FMCAD '02)*, volume 2517 of *LNCS*, pages 274–291. Springer Verlag, 2002.

[17] C.-J. H. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 24(9):1381–1405, Sept 2005.

Automatic Generation of Schedulings for Improving the Test Coverage of Systems-on-a-Chip

C. Helmstetter*† , F. Maraninchi*, L. Maillet-Contoz† and M. Moy*
*Verimag, Centre équation - 2, avenue de Vignate,
38610 GIÈRES — France
†STMicroelectronics, HPC, System Platform Group.
850 rue Jean Monnet, 38920 CROLLES — France

***Abstract*— SystemC is becoming a de-facto standard for the early simulation of Systems-on-a-chip (SoCs). It is a parallel language with a scheduler. Testing a SoC written in SystemC implies that we *execute* it, for some well chosen data. We are bound to use a particular deterministic implementation of the scheduler, whose specification is *non-deterministic*. Consequently, we may fail to discover bugs that would have appeared using another valid implementation of the scheduler. Current methods for testings SoCs concentrate on the generation of the inputs, and do not address this problem at all. We assume that the selection of relevant data is already done, and we generate several schedulings allowed by the scheduler specification. We use dynamic partial-order reduction techniques to avoid the generation of two schedulings that have the same effect on the system's behavior. Exploring alternative schedulings during testing is a way of guaranteeing that the SoC description, and in particular the embedded software, is scheduler-independent, hence more robust. The technique extends to the exploration of other non-fully specified aspects of SoC descriptions, like timing.**

I. Introduction

The Register Transfer Level (RTL) used to be the entry point of the design flow of hardware systems, but the simulation environments for such models do not scale up well. Developing and debugging embedded software for these low level models before getting the physical chip from the factory is no longer possible at a reasonable cost. New abstraction levels, such as the *Transaction Level Model (TLM)* [1], have emerged. The TLM approach uses a component-based approach, in which hardware blocks are modules communicating with so-called *transactions*. The TLM models are used for early development of the embedded software, because the high level of abstraction allows a fast simulation. This new abstraction level comes with new synchronization mechanisms which often make existing methods for RTL validation inapplicable. In particular, recent TLM models do not have clock anymore.

SystemC is a C++ library used for the description of SoCs at different levels of abstraction, from cycle accurate to purely functional models. It comes with a simulation environment, and is becoming a *de facto* standard. As TLM models appear first in the design flow, they become reference models for SoCs. In particular, the software that is validated with the TLM model should remain unchanged in the final SoC. Here, we concentrate on testing methods for SoCs written in SystemC.

The current industrial methodology for testing SoCs in SystemC is the following. First, we identify what we want to test (the *System Under Test*, or SUT), which is usually an open system. We make it closed by plugging *input generators* and a *result checker*, called *oracle*. SCV [2] is a testing tool for SystemC. It helps in writing input generators by providing C++ macros for expressing constraints: `SCV_CONSTRAINT((addr()>10 && addr()< 50)|| (addr()>=2 && addr()<= 5));` is an SCV constraint that will generate random values of `addr`. In most existing approaches, the SUT writes in memory, and the oracle consists in comparing the final state of the SUT memory to a reference memory. As usual, the main difficulty is to get a good quality test suite, i.e., a test suite that does not omit *useful* tests (that may reveal a bug) and at the same time avoids *redundant* tests (that can expose the same bugs) as much as possible. Specman [3] is a commercial alternative of SCV which uses the *e* language for describing the constraints.

Contributions and Structure of the paper: We assume that the choice of relevant data for the testing phase has already been done: we consider a SoC written in SystemC, including the data generator and the oracle. For each of the test data, the system has to be *run*, necessarily with a particular *implementation* of the scheduler. Since the *specification* of the scheduler is non-deterministic, this means that the execution of tests may hide bugs that would have appeared with another valid implementation of the scheduler. Moreover, the scheduling is due to the simulation engine only, and is unlikely to represent anything concrete on the final SoC where we have true parallelism. We would like the SoC description, and in particular the embedded software, to be scheduler-independent. Exploring alternative schedulings is a way of validating this property.

We present an automatic technique for the exploration of schedulings in the case of SystemC. It is an adaptation and application of the method for *dynamic* partial order reduction presented in [4]. This method allows to explore efficiently the states of a system made of parallel processes (given as object code) that execute on a preemptive OS and synchronize with a lock mechanism. We show here that it can be applied to SystemC too. Adaptations are needed because: the SystemC scheduler is not preemptive; SystemC programs use non-persistent event notifications instead of locks; evaluation phases alternate with update phases; an eligible process cannot be disabled by another one.

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

Our tool is based on forking executions: we start executing the system for a given data-input, and as soon as we suspect that several scheduler choices could cause distinct behaviors, we fork the execution. We use an *approximate* criterion to decide whether to fork executions. The idea is to look at the actions performed by the processes, in order to guess whether a change in their order (as what would be produced by distinct scheduler choices) could affect the final state. This criterion is approximate in the following sense: we may distinguish between executions that in fact lead to the same final state; but we cannot consider as equivalent two executions that lead to distinct final states. The result is a complete, but not always minimal, exploration of the scheduling choices for the whole data-input.

The paper is structured as follows: section II presents an overview of SystemC. Section III is the formal setting; Section IV explains the algorithms and section V proves the properties of the method. We present our implementation and evaluate it in section VI, related work in section VII, and we conclude with section VIII.

II. SystemC and the Scheduling Problems

A TLM model written in SystemC is based on an *architecture*, i.e. a set of components and connections between them. Components behave *in parallel*. Each component has typed connection *ports*, and its behavior is given by a set of communicating *processes* that can be programmed in full C++. For managing the set of concurrent processes that appear in the components, SystemC provides a *scheduler*, and several synchronization mechanisms: the low-level *events*, the synchronous *signals* that trigger an event when their value changes, and higher level, user-defined mechanisms based on abstract communication channels.

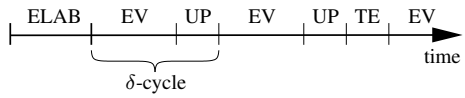


Fig. 1. Diagram of an execution

The static architecture is built by executing the so-called *elaboration phase* (ELAB), which creates components and connections. Then the scheduler starts running the processes of the components, according to the informal automaton of figure 2. Simulations of a SystemC model look like sequences of *evaluation phases* (EV). Signals *update phase* (UP) and *time elapse* (TE) separate them (see figure 1).

A. The SystemC Scheduler

According to the SystemC Language Reference Manual [5], the scheduler must behave as follows. At the end of the elaboration phase **ELAB**, some processes are *eligible*, some others are *waiting*. During the evaluation phase **EV**, eligible processes are run in an *unspecified order*, *non-preemptively*, and explicitly suspend themselves when reaching a *wait* instruction. There are two kinds of *wait* instructions: a process may wait for some time to elapse, or for an event to occur.

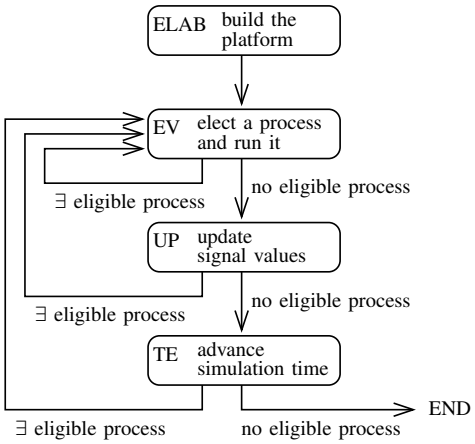


Fig. 2. Automaton of the SystemC Scheduler

While running, it may access shared variables and signals, enable other processes by notifying events, or program delayed notifications. An eligible process cannot become "waiting" without being executed. When there is no more eligible process, signals values are updated (**UP**) and δ-delayed notifications are triggered, which can wake up processes. A δ-cycle is the duration between two update phases. Since there is no interaction between processes during the update phase, the order of the updates has no consequence. When there is still no eligible process at the end of an update phase, the scheduler lets time elapse (**TE**), and awakes the processes that have the earliest deadline. A notification of a SystemC event can be immediate, δ-delayed or time-delayed. Processes can thus be become eligible at any of the three steps EV, UP or TE.

B. Examples

```
void top::A() {                  | void top::B() {
 wait(e);                        |  e.notify();
 wait(20,SC_NS);                 |  x = 0;
 if (x) cout << "Ok\n";          |  wait(20,SC_NS);
 else cout << "Ko\n";}           |  x = 1;}
```

Fig. 3. The `foo` example

To illustrate possible consequences of scheduling choices, let us introduce two small examples of SystemC programs. Figure 3 shows the example `foo` made of two processes `A` and `B`. It has three possible executions according to the chosen scheduling, leading to very different results:

- A;B;A;[TE];B;A: This scheduling leads to the printing of the string "Ok".
- A;B;A;[TE];A;B: The string "Ko" is printed. It is a typical case of *data-race*: `x` is tested before it has been set to 1.
- B;A;[TE];B: The execution ends after three steps only. The "`wait(e)`" statement has been executed before any notification of event `e`. Since events are not persistent in SystemC, process `A` has not been woken up. It is a particular form of *deadlock*.

```
void top::A()          | void top::C() {
  as in example foo    |    sc_time T(20,SC_NS);
void top::B()          |    wait(T);
  as in example foo    | }
```

Fig. 4. The `foobar` example

It is useful to test all executions of the `foo` example because they lead to different final states. But consider now the `foobar` example defined in figure 4. `foobar` has 30 possible executions, but only 3 different final states. 12 executions are equivalent to "C;A;B;A;[TE];C;B;A", 12 to "C;A;B;A;[TE];C;A;B" and 6 to "C;B;A;[TE];C;B". The method we present generates only 3 executions, one for each final state (or equivalence class).

In general testing techniques, the idea of generating one representative in each class of an equivalence relation is called *partition-based testing* [6]. It is not always formally defined.

C. Communication Actions

We call *communication actions* all actions that affect or use a shared object. We consider only two kinds of shared objects: events and variables. All other synchronization structures can be modeled using these two primitives.

There are two operations on events: *wait* and *notify*; and two operations on variables: *read* and *write*. In the sequel we will distinguish *caught* notifications (those that have woken up a process) from *missed* notifications, and *writes* that have modified the current value from non-modifying ones. Of course, theses distinctions can only be done dynamically in the general case.

III. Formal Setting

We will now explain how we generate schedulings for multi-threaded models written in SystemC. In the whole section, the SUT is a SystemC program. We suppose that we have an independent tool for generating test cases that only contain the data. We call SUTD the object made of the SUT plus one particular test data[1]. We have to generate a relevant set of schedulings for this data.

Most of the definitions in this section are quite standard in the literature on partial order reduction techniques.

A. Representation of the SUTD

When data is fixed, a SUT execution is entirely defined by its scheduling; a scheduling is entirely defined by an element of $(P \cup \{\delta, \chi\})^*$ where P is a process identifier and δ, χ are special symbols used to mark the δ-cycle changes and time elapses respectively. We consider full states of a SUTD to be full dumps of the SUTD memory, including the position in the code of each process. The SUTD can be seen as a *function* from the schedulings to the full states. It is partial: not all the elements of $(P \cup \{\delta, \chi\})^*$ represent possible schedulings of the SUTD (because of the synchronization constraints between processes).

[1]Strictly speaking, the SUT includes a data generator, not a single piece of data. But the generator does not depend on the scheduling, hence the distinction is not necessary here.

Definition 1 (Schedulings): Let M be a SUTD. P_M is the set of its processes; S_M is the set of its reachable full states; $F_M : (P_M \cup \{\delta, \chi\})^* \longrightarrow S_M$ is its associated *function*. F_M is partial. A *scheduling* is an element of $(P_M \cup \{\delta, \chi\})^*$; a *valid scheduling* is an element of the definition domain of F_M: $D_{F_M} \subset (P_M \cup \{\delta, \chi\})^*$.

For the programs of Section II-B, we have: $D_{F_{\texttt{foo}}} = \{ABA\chi BA, ABA\chi AB, BA\chi B\}$ and $F_{\texttt{foobar}}(ABC) = F_{\texttt{foobar}}(ACB) = F_{\texttt{foobar}}(CAB)$.

Definition 2 (Transitions): A *transition* is one execution of one process in a particular scheduling. Each transition of a scheduling is identified by its process identifier indexed by the occurrence number of this process identifier in the scheduling. For example, in the scheduling pqp there are 3 transitions: p_1, q_1 and p_2, in that order.

Definition 3 (Permutations): Let $u = vp_iwq_j$ be a valid scheduling where the transition p_i (resp. q_j) corresponds to the i-th (resp. j-th) execution of process p (resp. q). *Permuting the transitions p_i and q_j* means generating a new valid scheduling u' such that u' begins by v and the j-th transition of q in u' is before the i-th transition of p: there exists x, y, z such that $u' = vxq_jyp_iz$. u' is called a *permutation of p_i and q_j* for u.

We will use letters p, q, r to denote processes, $a, b, c, \ldots$ to denote transitions and $u, v, \ldots$ to denote sub-sequences of schedulings. Indexes will be omitted when obvious by context. An equivalence on the set of schedulings is needed to determine whether two schedulings lead to the same final state. We first define the relation $\sim$:

$\forall uabv \in D_{F_M},\ uabv \sim ubav \Leftrightarrow$
$\quad (ubav \in D_{F_M} \wedge F_M(uabv) = F_M(ubav))$

Definition 4 (Equivalence of Schedulings): The equivalence of schedulings is the reflexive and transitive closure of the relation $\sim$. It is noted $\equiv$.

This definition complies with the property: $\forall u, v \in D_{F_M}, u \equiv v \Rightarrow F(u) = F(v)$. Therefore, if we generate one element of each equivalence class of $\equiv$, we will have all possible final states. It allows to detect all property violation as soon as the corresponding output checker has been included into the SUT and drives it to a special final state when it detects an error.

B. Transition Dependency and Permutation Choice

We produce alternative schedulings by permuting some transitions of a given scheduling, but only when this can lead to a non-equivalent scheduling. For example, suppose that we are executing a SUTD and we have just executed the process p and then the process q ($u = u_1p_iq_j$). If there is no causal reason why the transition q_j was after the transition p_i (process q was not waiting for an event notified in p_i), then we can permute these two transitions. In that case, executing q instead of p in the state $F_M(u_1)$ can be a divergent path as illustrated on figure 5. The question we have to answer is: "Do these two schedulings lead to the same state?" or formally: "$F_M(u_1pq) = F_M(u_1qp)$?". Note that we may not be able to prove that $F_M(u_1pq) = F_M(u_1qp)$ because we want to answer this question *without* executing u_1qp entirely. Hence we rely on the common objects accessed by the transitions to

guess whether a permutation has some effect on the final state. This is incomplete. If we cannot prove that the final states are equal, we generate the new scheduling.

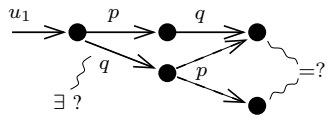


Fig. 5. A Potential Divergent Path, black circles represent global states of the model

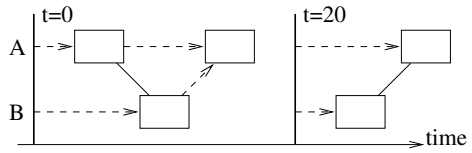


Fig. 6. Dynamic Dependency Graph

We now study the two questions: which transitions *can* we permute? which transition permutations are *useful*? The answer to the first question is given by the *permutability relationship*; the answer to the second question is given by the *commutativity* relationship (it is useless to permute commutative transitions).

The *Dynamic Dependency Graph (DDG)* represents the synchronizations that occur for a particular scheduling. Figure 6 represents the scheduling $aba\chi ba$ of the `foo` program of figure 3. Each horizontal line is a process. New cycles (δ or χ) are represented by vertical lines. Each box is a process transition. Dashed arrows (resp. plain lines) between boxes indicate that the two transitions are dependent but not permutable (resp. non commutative). We may move some transitions on the horizontal axis, remaining among the *valid and equivalent schedulings*, provided we do not permute two boxes linked by an arrow or line.

Definition 5 (Permutability): The transitions a and b are *causally permutable* in the valid scheduling $u_1au_2bu_3$, noted $(a,b) \in P$, if and only if: $\{u_1v_1ba \in D_{F_M} | \exists v_2, u_1v_1abv_2 \equiv u_1au_2bu_3\} = \emptyset$.

In other words, two transitions are not permutable if:

1) there is an equivalent scheduling in which they are consecutive;
2) the second transition b can be elected in place of the first transition a in this equivalent scheduling.

Definition 6 (Commutativity of Transitions): The non-causally ordered transitions a and b are *commutative* in the valid scheduling $u_1au_2bu_3$ if and only if:

$\forall u_1v_1abv_2 \equiv u_1au_2bu_3, u_1v_1abv_2 \equiv u_1v_1bav_2$

Commutativity is not defined for causally ordered transitions.

The theory of partial order reduction relies on the definition of *dependent* transitions [7]. In our work, we define the dependency relationship D as follows:

Definition 7 (Dependency of Transitions): The transitions a and b are *dependent* if and only if they are not permutable, or permutable but not commutative.

The *causal order* specifies which transitions can be permuted in a particular scheduling without permuting dependent transitions, including themselves. All schedulings of the same equivalence class have the same causal order. Unlike the permutability relationship, the causal order is a partial order.

Definition 8 (Causal Order): The transitions a and b are *causally ordered* in the valid scheduling $u = u_1au_2bu_3$, noted $a \prec_u b$, if and only if $(a,b) \in$ transitive closure of $\{(x,y) \in D | x <_u y\}$.

IV. Algorithms

A. Computation of the Commutativity Relationship

The first step is to detect pairs of transitions which are not commutative. We compute here a relationship C for all pairs of transitions. This computed relationship is correct for permutable transitions, which is sufficient for our problem. Two transitions may be non-commutative ($(a,b) \notin C$) only if they contain non-commutative *communication actions* on the same shared object (see section II-C). Note that the order of these actions within a transition is irrelevant. We examine all cases below.

For shared variables there are three cases of non-commutative actions (since operations on variables have no effect on process eligibility, we just need to check whether the equality of resulting states is still verified after permutation):

1) a *read* followed by a *modifying write*
2) a *modifying write* followed by a *read*
3) a *write* followed by a *modifying write*

In all other cases, the transitions are commutative, as in example 2. Note that the nature of a *write* depends on the scheduling we consider. A *modifying write* can become a *non-modifying write* for another scheduling, and reciprocally.

Example 1: Variable `x` initially set to 0. The first transition executes the action `x=x+2`. The second executes `x=4-x`. It is a *modifying write* followed by a *read* so we consider that the two transitions are not commutative (point 2 above).

Example 2: Variable `x` initially set to 2. The first transition executes the action `x=4`. The second transition also executes this instruction. It is a *modifying write* followed by a *non-modifying write*.

Note that C is symmetric, which may not be obvious from point 3 above. But permutating a *modifying write* with a *non-modifying write* is still a *modifying write* followed by a *non-modifying write*, except if there is another pair of dependent actions. Example 2 also illustrates this remark.

For events, there are three cases of non-commutative actions:

1) a *notification* followed by a *wait*
2) a *wait* followed by a *notification*
3) a *caught notification* followed by a *notification*

The dependency between a *wait* and a *notify* is quite obvious: if the *wait* comes first, then the corresponding process is woken up by the *notify*, otherwise it remains sleeping. Example 3 illustrates the third case.

Example 3: Suppose one runs this three-process model:

- Initial state: process `A` waiting for `e`, `B` and `C` eligible.
- Process `A`: `cout <<'a'; x = 1;`

- Process B: `cout <<'b'; x = 2; e.notify();`
- Process C: `cout <<'c'; e.notify();`

There is exactly one transition per process, noted a, b and c. Four schedulings are valid: bac, bca, cba and cab. In bac and bca, b is dependent with a (2 *modifying writes*) but they are causally ordered (process A was enabled by the transition b). However if we permute b and c, b is no longer causally ordered with a since A was enabled by c instead of b.

Permuting two notifications of an event does not modify the resulting state of the SUTD, but modifies the computed causal order. That's why they are considered as non-commutative.

B. Computation of the Causal Partial Order

In order to compute the permutability, we need to compute the causal order $\prec$. We denote prec(u) the set $\{a, b \in u | a \prec b\}$ obtained after the execution of the scheduling u.

We compute the causal order step by step. Obviously, for the empty scheduling we have prec$(\epsilon) = \emptyset$. Let a and b be two transitions, we have $a \prec b$ and so $(a, b) \in D$ at least in the three following cases:

- a or b indicate a new δ-cycle or time-elapsed.
- a and b belong to the same process (by definition)
- the process of transition b has been woken up by a.

In these cases, we note: $a \prec_\beta b$. The rest of the paragraph below is adapted from [4]. Having prec(u), we compute prec(ub) as follows:

$$\begin{aligned} \text{prec}_1(ub) &= \text{prec}(u) \cup \{a \prec_\beta b | a \in u\} \\ \text{prec}_2(ub) &= \text{prec}_1(ub) \cup \{(a, b) \notin C | a \in u\} \\ \text{prec}(ub) &= \text{transitive closure of prec}_2(ub) \end{aligned}$$

Finally, we have $(a, b) \in P$ in $u_1 a u_2 b u_3$ if and only if: $(a, b) \in$ transitive closure of prec$_1(u_1 a u_2 b)$.

The following property is useful to optimize the implementation: Let $u_1 a u_2 b u_3 c u_4$ be a scheduling. Then process$(a) =$ process$(b) \wedge b \prec c \Rightarrow a \prec c$. Owing to this property, we can represents the causal order with an array T of size $p \times s$ where s is the number of steps and p is the number of processes. The element $T[a, q]$ is the last transition of process q which is causally before a; i.e.: $a \prec b \Leftrightarrow$ num$(a) \leq T[b,$ process(a)$]$. Some other optimizations are well explained in [4].

C. Generation of one alternative scheduling

We are now able to determine if two transitions are not commutative (hence should be permuted). Now we explain how we treat such a pair of transitions. Let $uavb$ be a scheduling such that $(a, b) \in D \cap P$. Let $v = v_1 \dots v_n$ where $v_1, \dots, v_n$ are transitions. The goal is to generate a new valid scheduling with b before a. We proceed as follows:

- The first part u is unmodified.
- We execute all v_i such that $a \not\prec v_i$.
- We execute b and then a (unlike some other concurrent languages, b cannot disable a in SystemC).
- Then, since two dependent transitions have been permuted, we do not know whether the non-executed transitions v_i such that $a \prec v_i$ are still defined. We are then free to choose the rest of the scheduling.

D. Generation of a full schedulings suite

We start by executing the SUTD with a random scheduling. In parallel with the SUTD execution, we run a checker:

- the checker computes the causal partial order "$\prec$" and builds the Dynamic Dependency Graph.
- if it discovers two non-commutative transitions p_i and q_j, with p_i before q_j:
  - it generates a new scheduling such that q_j before p_i by permuting the transitions with the algorithm described above; the constraint "q_j before p_i" is saved with the new scheduling to prevent further permutations of the same transitions.
  - it continues the current execution, adding the opposite constraint "p_i before q_j" to all of its further children.

Then we replay the SUTD with each generated scheduling u. When we reach the end of u, we continue the SUTD execution with a random scheduling. In parallel, we compute the causal order and generate new schedulings for each non-commutative pair of transitions, as for the previous schedulings. Thanks to the constraints saved with the generated schedulings, each new generated scheduling is more constrained than its father scheduling and so there are fewer and fewer new schedulings at each iteration. When the checker does not generate any new scheduling, we have a complete test suite.

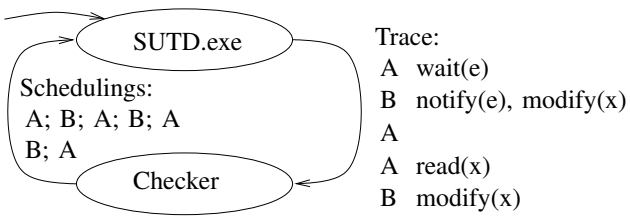


Fig. 7. First iteration of the analysis for the foo example. The first execution activates processes A and B in the order $ABAAB$. The checker generates two new schedulings. One to permute A_1 and B_1 (unordered accesses to event e) and the other to permute A_3 and B_2 (unordered accesses to shared variable x).

V. Properties

The algorithm guarantees that we generate at least one element of each equivalence class (for the equivalence of definition 4).

Theorem 1: Let G_M be the set of all generated schedulings of a model M. For any scheduling $u \in D_{F_M}$, there exists a scheduling $v \in G_M$ such that $u \equiv v$.

There are two useful and direct corollaries. First, if a local process state is present in a scheduling of D_{F_M}, it is also present in a scheduling of G_M. Furthermore, we generate all the final states, including all deadlocks.

To prove the property, we need the definition of $\equiv-$prefix and $\equiv-$dominant for schedulings, directly adapted from *prefix* and *dominant* properties of Mazurkiewicz traces [7].

Definition 9: Let $p, d \in D_{F_M}$ be two schedulings, p is an $\equiv-$prefix of d and d an $\equiv-$dominant of p if and only if there exists a scheduling $u \in D_{F_M}$ such that $u \equiv d$ and p is a string-prefix of u.

Proof: We proceed by contradiction, and assume that there exists a scheduling $u \in D_{F_M}$ which breaks the property. We can write u in the form $u = u_1 a u_2$ where u_1 is the longest prefix of u such that:

$$\exists u_1 u_2' \in D_{F_M} \text{ and } v \in G \text{ such that } u_1 u_2' \equiv v$$

This decomposition is unique so we just have to prove that $u_1 a$ has an $\equiv-$dominant in G to get the wanted contradiction.

Let $v \in G$ be a generated completed scheduling such that u_1 is a $\equiv-$prefix of v. As a consequence, there exists a valid scheduling $u_1 u_2'$ such that $u_1 u_2' \equiv v$ If there is no non-determinism when we are in the state $F_M(u_1)$, then we must have $u_2' = a u_3'$ and so v would be a $\equiv-$dominant of $u_1 a$.

Consequently a is neither δ nor χ and the process of a is defined and eligible in $F_M(u)$. Since an eligible process cannot become "sleeping" without running, a is present in u_2' so $u_2' = w_1 a w_2$. Since a is eligible in $F_M(u)$, it is not causally after any element of w_1. There are three cases:

- if w_1 is empty then we get the needed contradiction
- if $w_1 = xb$ with b I a then there exists another possible scheduling $u_1 u_2'' \equiv v$ such that $u_2'' = w_1' a b w_2$ with w_1' shorter than w_1.
- if $w_1 = xb$ with $(b, a) \in D$ then:
  - Transition b is before a in v but they are permutable.
  - So we have generated a scheduling v' with a before b, using the algorithm described in section IV-C.
  - There exists a possible scheduling $u_1 u_2'' \equiv v'$ such as $u_2'' = w_1' a b w_2'$ with w_1' shorter than w_1.

Consequently, by induction on the length of w_1, we get the needed contradiction. ■

VI. Prototype Implementation and Evaluation

A. The prototype

Figure 8 is an overview of the tool. The **checker** implements the checking algorithm of section IV-D. It has to be aware of all communication actions. Some of them can be detected by **instrumenting** the SystemC kernel, some other cannot (like accesses to a shared variable, that are invisible from the SystemC kernel). We choose to instrument the C++/SystemC source code. For each communication action in the code of a SystemC process, we add an instruction that notifies the operation to a global recorder. For example, consider the instruction `x=y` where `x` and `y` are shared variables. The two following instructions are added close to the assignment: `recorder->read(&y);recorder->write(&x)`. Instrumentation is based on the open-source SystemC front-end Pinapa [8], and is compositional.

Another solution would have been to interpret or instrument the binaries. However, using a SystemC front-end has some benefits: it allows to generate a *static dependency graph (SDG)* which represents a superset of the communications that can occur between processes (see Figure 9). Moreover, it is easier to link the observed behavior to the source code.

The instrumented SystemC program is compiled with a **patched SystemC kernel**. The patches are: 1) replacing the election algorithm of the SystemC scheduler by an interactive version, still complying with the SystemC specification;

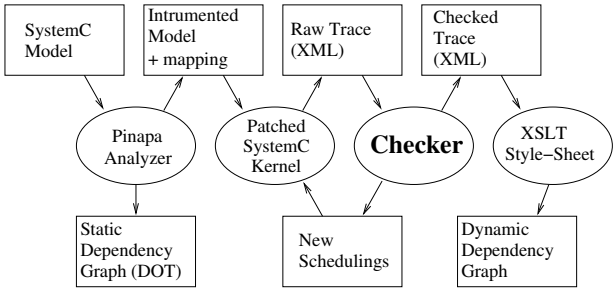


Fig. 8. The Prototype's Architecture

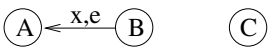


Fig. 9. Static Dependency Graph for the `foobar` example. Nodes represent processes. Arrows represent possible communications between processes. An arrow goes from the master (i.e. the notifier for a SystemC event, the writer for a shared variable) to the slave.

2) adding code to record the communication actions that cannot be detected in the code of the processes, and their consequences (e.g., enabling of a process). When we execute the instrumented platform with the patched SystemC kernel, we can detect dependencies dynamically or save a detailed trace and run the checker afterwards. In both cases, we get a list of new schedulings to be executed, and a record of the computed dependencies, usable as input for other checkers or visualization tools, like the production of the dynamic dependency graph (DDG).

B. Evaluation

In order to validate our tool and to evaluate the quality of the test suites produced, we studied several industrial SoC models. Assume that running one test-case takes some time T. In order to cover the scheduling choices, we have to run more than one test-case. Let us denote V the number of *valid* schedulings, and G the number of schedulings *generated* by our tool. It is interesting to compare $V \times T$ with $G \times T + O$, where O is the overhead due to the computation of new schedulings.

With a real application, it is often difficult to evaluate V. We chose to evaluate our method on three examples. First, we considered a SystemC encoding of the indexer problem presented in [4][2], because it is easy to evaluate V. However, the indexer is not representative of the typical SystemC code found in industry. We then looked at two industrial case-studies: the first one has about 50 000 lines of code but only 4 processes, and it does not model a full SoC; the second one has about 250 000 lines of code and 57 processes, and it represents a full SoC.

1) The Indexer Example: There are n components and one global 128-element array used as a hash table. Each component is composed of 2 threads which communicate using a shared variable and a SystemC event. Each component writes 4 messages in the global hash table. This corresponds to schedulings of length $11 \times n$. For $n \leq 11$, there is no collision

[2]For the SystemC version see: `http://www-verimag.imag.fr/~helmstet/indexer.cpp`.

in the hash table and all schedulings lead to the same final state. For $n \geq 12$ there are collisions hence non-equivalent schedulings. Our prototype generates valid schedulings leading to distinct states of the hash table. In this example, we generate exactly one scheduling per equivalence class. The number of generated schedulings is far smaller than the number of valid schedulings (at least $3.35E11$ for $n = 2$, and $2.43E25$ for $n = 3$). Results are summarized in table I. Time is given only to help estimating the curve, not as an absolute measure.

components	generated schedulings	time
1 . . . 11	1	$\leq$ 11 ms
12	8	60 ms
13	64	4 s
14	512	35 s
15	4096	5 mn

TABLE I
RESULTS FOR THE INDEXER EXAMPLE

2) The MPEG Decoder System: This system has 5 components: a master, a MPEG decoder, a display, a memory and a bus model. There are about 50 000 lines of code and only 4 processes. This is quite common in the more abstract models found in industry, because there is a lot of sequential code, and very few synchronizations. We added 340 instrumentation lines to detect communication actions.

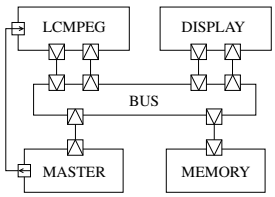


Fig. 10. Architecture of the MPEG decoder system

The test is stopped after the third decoded image, which corresponds to 150 transitions. One simulation takes 0.39 s. Our tool generates **128 schedulings** in **1 mn 08 s**. No bug is found, which guarantees that this test-case will run correctly on any SystemC implementation. Running the model 128 times takes more time than generating the schedulings (we have $G \times T = 128 \times 0.39$ s ≈ 50 s and $O \approx 1$ mn $08 - 50$ s ≈ 18 s). Thus the overhead O remains acceptable.

On this example, we noticed that the number of generated schedulings could be improved. This MPEG decoder, as many other TLM models, uses a pair (event, variable) to implement a *persistent event* as follows (`x` is initially 0):

Process `P` runs: `x=1; e.notify();`
Process `Q` runs: `if (!x) wait(e); x=0;`

The two valid schedulings `P; Q` and `Q; P; Q` lead to the same final state, but our tool currently generates both schedulings because it cannot prove it. The intuition is that these schedulings are not equivalent according to the dependency relationship as computed in section IV. Detecting this kind of structures in the source code and taking them into account for the computation of the dependency relationship would allow to generate less schedulings.

3) A Complete SoC: Complete models of Socs are typically 3 to 6 times bigger than the MPEG decoder. We are currently evaluating our tool on a model —let us call it XX— corresponding to a full SoC: it has about 250 000 lines of code and 57 processes. At the moment we are limited by the code instrumentation tool which still requires some manual work, so we looked at only one case study of this type, but the instrumentation tool will soon be fully automatic. For tests of length around 200 transitions, we expect the tool to behave well on XX: the ability to cope with this number of processes has been tested with the indexer example, and the ability to cope with the complexity of a large and realistic SystemC description has been tested with the MPEG example.

The interesting point with XX is the *granularity* of the transactions. With the MPEG decoder, the granularity corresponds to an algorithm that takes one line of the image at a time. Something interesting can be observed by a test oracle after 150 transitions only (three images have already been decoded). XX corresponds to an algorithm that takes one pixel of the image at a time. It may be the case that the test oracle has to observe thousands of transitions. XX is a very good case-study for observing the combined influence of the test length and the granularity on the performances of our technique. One phenomenon we can expect, and that we have to validate with the case-study, is the following: very abstract TLM descriptions have large-grain transactions, but loose synchronisations; while the more detailed TLM descriptions have finer-grain transactions, but stronger synchronizations. If the number of alternative schedulings decreases (because of stronger synchronizations) when the granularity of a description increases (and thus the length of the interesting test-cases), the method may still be applicable. We also comment on this point in the conclusion.

VII. RELATED WORK

Existing work (see, for instance [9]) addresses formal verification for TLM models. The idea is to extract a formal model from the SystemC code, and to translate it into the input format of some model-checker. In such an approach, the complete model that is model-checked has to include a representation of the scheduler. It is sufficient to use a non-deterministic representation that reflects the specification of SystemC, and then a property that is proved with this non-deterministic scheduler is indeed true for any deterministic implementation. Model-checking is likely to face the state-explosion problem, so testing methods are still useful. But we need the same guarantee on the results of the test being valid for any implementation of the simulation engine.

Partial order reduction techniques are quite old, but their *dynamic* extension is quite recent. As far as we know, it is not included in VERISOFT [10] yet. Partial order reduction is used in many model checkers for asynchronous concurrent programs such as Spin [11] or JAVA PATHFINDER [12]. However, since we use testing, our work is more related with tools which work directly on the program without abstractions,

such as VERISOFT or CMC [13]. The main difference is that our tool is adapted to the TLM SystemC constructs.

To get a complete validation environment, one need to include a test case generator and an output checker. For the latter, *assertion-based verification* [14] proposes to derive monitors from assertion languages. However, these languages are often based on the notion of clocks which are absent in TLM. If ABV is extended to TLM, it will become useful in our framework.

VIII. CONCLUSION AND FURTHER WORK

We presented a method to explore the set of valid schedulings of a SystemC program, for a given data input. This is necessary because the scheduling is a phenomenon due to the simulation engine only, and is unlikely to represent anything concrete on the final SoC. Exploring alternative schedulings during testing is a way of guaranteeing that the SoC description, and in particular the embedded software, is scheduler-independent, hence more robust. By using dynamic partial order reduction, we maximize the coverage and keep the number of tests as low as possible. Our tool also produces several graphical views that help in debugging SoCs. With the prototype tool, we have highlighted unwanted non-determinism in a bus arbiter for a transaction-accurate protocol. Also, some SoC descriptions are scheduler-dependent because they exploit the initial state of the most used implementation. In this case, covering the valid schedulings reveals deadlocks. Our tool is already mature enough to be used for industrial SystemC descriptions of SoCs.

There are at least two ways of improving the prototype performances. The first is to reduce the number of branches explored. A promising solution is to use partial state memorization. It is unrealistic to save all the states and compare the new state at each step due to the size and complexity of a SystemC model state. However, we can save some states and compare only particular new states. We plan to compare each forked execution every new delta-cycle. The second way is to reduce the time overhead needed for runtime checking. Some check results are predictable. Consequently doing static analysis before simulation can avoid runtime computation.

Further work on testing SoCs is threefold. First, the algorithm that fully explores alternative schedulings can be used on large platforms only if the length of the test is reasonable. A promising idea for very long tests is to use the method *locally* on the TLM description: a first execution of the whole platform P is used to record the output transactions of some sub-system S of P. Then, our method is applied on a platform P' obtained by substituting S' with S in P. S' is a sequential algorithm that plays the recorded transactions. It does not introduce scheduling choices. The idea is that the method then concentrates on the schedulings due to P$-$S, forgetting the schedulings due to S.

Second, the whole approach and the SystemC prototype is being adapted to the exploration of non-fully specified timings in the TLM models. Indeed, TLM models are not cycle-accurate, but people use to label them by approximate timing properties of the components, in order to estimate the timing properties of the SoC early. In this case, the timings should not be taken as fixed values. The embedded software will be more robust if it works correctly for slightly distinct timings. In the testing process, it is useful to explore alternative timings, with the same idea of generating only those timings that are likely to change the global behavior of the SoC. An overview of the method can be found in [15].

We also started working on efficient implementations of the SystemC simulation engine, by exploiting multi-processor machines. Here, the difficulty is to guarantee that a multi-processor simulation does not exhibit behaviors that are not allowed by the non-deterministic reference definition of the scheduler. The formal setting we described here is appropriate for defining the set of behaviors that the multi-processor simulation may produce, without changing the behavior of the embedded software.

REFERENCES

[1] F. Ghenassia, Ed., *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005, iSBN 0-387-26232-6.

[2] J. Rose and S. Swan, "SCV Randomization," Cadence Design Systems, Inc., 2003, `www.testbuilder.net/reports/scv_randomization.pdf`.

[3] T. Kuhn, T. Oppold, M. Winterholer, W. Rosenstiel, M. Edwards, and Y. Kashai, "A framework for object oriented hardware specification, verification, and synthesis," in *DAC '01: Proceedings of the 38th conference on Design automation*. New York, NY, USA: ACM Press, 2001, pp. 413–418.

[4] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Symposium on Principles of programming languages (POPL)*. New York, NY, USA: ACM Press, 2005, pp. 110–121.

[5] *SystemC v2.0.1 Language Reference Manual*, Open SystemC Initiative, 2003, `http://www.systemc.org/`.

[6] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proceedings of the international conference on Reliable software*, 1975, pp. 493–510.

[7] A. Mazurkiewicz, "Trace theory," in *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*. New York, NY, USA: Springer-Verlag New York, Inc., 1987, pp. 279–324.

[8] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "Pinapa," 2005, `http://greensocs.sourceforge.net/pinapa/`.

[9] M. Moy, F. Maraninchi, and L. Maillet-Contoz, "LusSy: A toolbox for the analysis of systems-on-a-chip at the transactional level," in *International Conference on Application of Concurrency to System Design*, June 2005.

[10] P. Godefroid, "Model checking for programming languages using VeriSoft," in *Symposium on Principles of Programming Languages (POPL)*, ACM, Ed. New York, NY, USA: ACM Press, 1997, pp. 174–186.

[11] G. J. Holzmann, "The model checker SPIN," *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[12] W. Visser, K. Havelund, G. Brat, and S.-J. Park, "Model checking programs," in *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[13] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[14] "Assertion-based verification," Synopsis, 2003, `http://www.synopsys.com/products/simulation/assertion_based_wp.html`.

[15] C. Helmstetter, F. Maraninchi, and L. Maillet-Contoz, "Test coverage for loose timing annotations," in *11th International Workshop on Formal Methods for Industrial Critical Systems*, August 2006.

SIMULATION BOUNDS FOR EQUIVALENCE VERIFICATION OF ARITHMETIC DATAPATHS WITH FINITE WORD-LENGTH OPERANDS *

Namrata Shekhar, Priyank Kalla
Electrical & Computer Engineering Department
University of Utah, Salt Lake City, UT-84112
{*shekhar, kalla*}*@eng.utah.edu*

M. Brandon Meredith, Florian Enescu
Department of Mathematics & Statistics
Georgia State University, Atlanta, GA-30303
{*fenescu@mathstat, mmeredith5@student*}*.gsu.edu*

Abstract— **This paper addresses simulation-based verification of high-level descriptions of arithmetic datapaths. Instances of such designs are commonly found in DSP for audio, video and multimedia applications, where the word-lengths of input/output bit-vectors are fixed according to the desired precision. Initial descriptions of such systems are usually specified as Matlab/C code. These are then automatically translated into behavioural/RTL descriptions (HDL) for subsequent hardware synthesis.**

In order to verify that the initial Matlab/C model is bit-true equivalent to the translated RTL, how many simulation vectors need to be applied? This paper explores results from number theory and commutative algebra to show that exhaustive simulation is not necessary for testing their equivalence. In particular, we derive an upper bound on the number of simulation vectors required to prove equivalence or identify bugs. These vectors cannot be arbitrarily generated; we determine exactly those vectors that need to be simulated. Extensive experiments are performed within practical CAD settings to demonstrate the validity and applicability of these results.

I. INTRODUCTION

Increasing size and complexity of digital systems has resulted in a vast array of formal verification techniques which operate at different levels of abstraction. In spite of many such advances, simulation-based validation has remained an important method for ensuring functional correctness during various stages of the design cycle. Fig. 1 describes a typical design flow for arithmetic datapath intensive (signal-processing) applications, along with the context in which the verification problem appears.

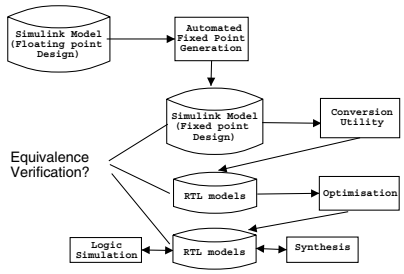


Fig. 1. The Equivalence Verification problem: Matlab to RTL design flow.

Initial algorithmic specifications (such as MATLAB models) of most signal processing applications involve data representation using floating-point formats. However, they are often implemented with fixed-point architectures, where the required precision dictates the bit-vector sizes of the variables. Various automated tools exist for this translation [1]. For synthesis and optimization purposes, these high-level descriptions may be subsequently converted to HDL using automatic utilities [2] [3]. Design optimization may be further achieved by applying high-level synthesis and restructuring operations on the translated RTL model [4] [5]. It is required to show that the translated and optimized RTL models are bit-true equivalent to the fixed-point specification.

Simulation is extensively used to validate the input-output behavior of the original model (at the MATLAB, C level); say to validate the pass-band of a filter. These vectors can then also be applied at the RT-level. Validation of the Matlab model is fast, even with a large number of test vectors, since it is compiled-code simulation. However, simulating the RTL model with the same set of vectors is generally slow. In this regard, this paper derives an important result related to simulation-based verification of high-level descriptions of arithmetic datapaths. In particular, we show that:

1. Exhaustive simulation is not always necessary to verify equivalence or to find bugs;
2. An upper bound is derived for the maximum number of test vectors required for this purpose; and
3. Which vectors to choose for simulation.

Further, we model such arithmetic-intensive design descriptions as polynomial functions. Moreever, the word-lengths of the variables are usually predetermined and fixed. For correct modeling, we need to account for the effect of the fixed bit-widths of the input/output variables. Hence, the polynomial functions are computed over finite-integer rings where the ring cardinality corresponds to the datapath size. We then apply concepts from number theory to systematically establish the claims mentioned above.

Let us motivate this issue using a practical example and put our contribution in perspective.

A. An Example Application

Given two degree-k polynomials $F_1(x)$ and $F_2(x)$, their coefficients can be represented as $(k+1)$-length vectors $A = (a_0, \ldots, a_k)$ and $B = (b_0, \ldots, b_k)$. Computing the *convolution* of such vectors results in another $k+1$ vector, according to:

$$c_i = \sum_{j=0}^{k} a_j b_{i-j} \quad 0 \leq i \leq k \tag{1}$$

where, c_i is the i^{th} component in the vector C. This procedure, however, has a complexity of $O(n^2)$. It is well-known [6] [7] that convolution can be effectively implemented in hardware by:

1. Computing the DFT of vectors A and B,

*This work is sponsored in part by NSF CAREER grant CCF-546859 and NSF grant CCF-515010.

2. Calculating their pairwise product; and finally,
3. Taking the inverse DFT of the result. In other words, the result vector $(c'_0, c'_1, \ldots, c'_k)$ is computed as

$$C' = DFT^{-1}(DFT(A) \cdot DFT(B)) \quad (2)$$

This operation is shown in Fig. 2 for degree-3 polynomials. Such a 'divide-and-conquer' strategy results in a complexity of $O(n \cdot logn)$, and is a popular way of implementing the convolution of two vectors.

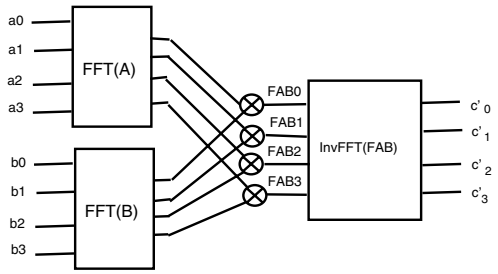


Fig. 2. Convolution of A and B.

Suppose that we intend to verify that both implementations compute the same values, say, for coefficients c_0 (obtained via direct convolution) and c'_0 (obtained via DFT-product-InvDFT). Further, as is often the case, assume that the entire datapath word-length in both cases is fixed to a certain width, say 4-bits. Since each of the inputs (a_i and b_i, $0 \leq i \leq 3$) can take values between $\{0, \ldots, 2^4 - 1\}$, is it necessary to simulate a total of 2^{4*8} test vectors to prove bit-true equivalence or to identify the presence of a bug?

This paper derives results which prove that in the above case: i) if the two designs are not equivalent (bug), a maximum of 6^8 simulations are sufficient to capture the erroneous behavior; ii) if for these 6^8 vectors, no bug is detected, then the designs are indeed equivalent. A method for generating these specific simulation vectors is also derived.

B. Problem Modeling and Scope

We model the arithmetic computations over bit-vectors as follows. Let $x_1, x_2, \ldots, x_d$ denote the d-variables (bit-vectors) in the design. Let $n_1, n_2, \ldots, n_d$ denote the size of the corresponding bit-vectors. Therefore, $x_1 \in Z_{2^{n_1}}, x_2 \in Z_{2^{n_2}}, \ldots, x_d \in Z_{2^{n_d}}$. Note that $Z_{2^{n_i}}$ corresponds to the finite set of integers $\{0, 1, \ldots, 2^{n_i} - 1\}$. Let m correspond to the size of the output bit-vector f; hence, $f \in Z_{2^m}$. Subsequently, we model the arithmetic datapath computation as a *polynomial function* (or polyfunction) from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m} [8]. Here $Z_a \times Z_b$ represents the *Cartesian product* of Z_a and Z_b. In other words, the computation is modeled as a multi-variate polynomial $F(x_1, x_2, \ldots, x_d)$ $\%2^m$, where each $x_i \in Z_{2^{n_i}}$ and F is computed $\%2^m$. The equivalence problem then corresponds to checking the congruence of two polynomials: $F \equiv G \%2^m$.

Note: Our approach is applicable to high-level descriptions of bit-vector arithmetic that can be easily abstracted as polyfunctions. For this reason, our approach cannot be used to verify a behavioural RTL model against its gate-level netlist.

The verification problem of Fig. 1 has seen a lot of interest recently in [9] [10] [11] [12] [13]. The works of [12] [13] use the same polynomial function model to derive a symbolic approach to prove/disprove equivalence of arithmetic datapaths. However, these works are restricted inasmuch as they can only provide a "yes/no" answer to the equivalence check. They cannot provide an error trace when bugs are detected. Moreover, our results also have implications on the applicability of the *fundamental theorem of algebra* which has been used in hardware design and verification [9] [10] [11], as discussed below.

C. Bit-Vector Arithmetic versus the Fundamental Theorem of Algebra

Lemma I.1: Let $P(x)$ be a degree-k uni-variate polynomial. If $P(x) = 0$ for $(k+1)$ distinct values of x, then all the coefficients of $P(x)$ are zero.

The above lemma [14] is based on the *fundamental theorem of algebra* [15]. This theorem states that a degree-k univariate polynomial $P(x)$ has exactly k complex roots, unless all its coefficients are zero. Since integers are a special case of complex numbers, this theorem holds for the set Z as well. The work of [10] used this result for equivalence verification by modeling arithmetic datapaths as polynomials, and showed that $k+1$ vectors were sufficient to prove equivalence of any given degree-k polynomials. It was later extended in [14] to be applicable to multi-variate polynomials as well, and was further applied to reduce the complexity of model-checking. Also, [9] used the same concepts for polynomial extraction for high-level synthesis purposes.

However, the above results are only relevant in unique factorization domains (UFDs), such as the set of real numbers (R), the set of integers (Z), finite fields ($Z_p, GF(p^n), p =$prime) and so on. Finite-word-length arithmetic does not correspond to UFDs. Since a bit-vector of size m represents integer values reduced $\%2^m$, bit-vector arithmetic corresponds to a ring (and not a field) Z_{2^m}, which contains zero-divisors. Therefore, when the bit-vector sizes are accounted for in our model, polynomials are computed modulo an integer power of 2. Consequently, factorization is not unique as shown for the polynomial $F(x) = x^2 + 6x$ below.

$$\begin{aligned} x^2 + 6x &= x(x+6) \ \ \%2^3 \\ &= (x+4)(x+2) \ \ \%2^3 \end{aligned}$$

The polynomial $F(x)$ has a degree $k = 2$, but can be factorized in two non-unique ways; corresponding to four unique roots. In such cases, Lemma I.1 does not hold; simulating for $k+1$ values such as $x = 0, 2, 4$, does show $F = 0$ but that does not mean that all the coefficients of $F(x)$ are zero. Therefore, simulating for only $k+1$ vectors is insufficient for verification.

Clearly, for finite-word-length bit-vector arithmetic, properties of these class of rings (of the type Z_{2^m}) need to be investigated further for simulation-based verification. This paper explores results for polynomial functions over such finite integer rings and develops solutions to such problems with applications in simulation-based verification of arithmetic datapaths.

D. Paper Organization

This paper is organized as follows: The next section reviews related work in simulation-based verification. Section III covers preliminary concepts and background material regarding polynomial functions and finite ring theory. Section IV describes the proposed results for univariate polynomials and provides the mathematical foundation for their support. These results are extended for multi-variate computations as well. Finally, Section V describes the experimental setup and results, while Section VI concludes the paper.

II. Previous Work

Bryant, in [16] [17], used three-valued logic simulation to reduce the required number of simulation vectors for circuit verification. Subsequently, [18] incorporated some of these techniques into their framework, which provided a method for design verification at different levels of abstraction. Brand [19] proposed exploiting information from the design specification to significantly reduce the complexity of simulation. Clarke *et al.* further researched the problem of specifications and generators in [20]. However, BDDs were used to demonstrate a practical approach to this problem in the *SimGen* project [21]. Later on, Shimizu *et al.* [22] [23] automated this approach to verify large industrial designs. The above methods are focused towards generating the appropriate number of test vectors to ensure sufficient verification coverage. Our approach, on the other hand, applies polynomial methods that make exhaustive simulation unnecessary for arithmetic datapaths.

An algebraic approach to reducing the test vector set was proposed in [10], and later extended in [11]. These works model the given datapaths as polynomials and apply the fundamental theorem of algebra to verify the descriptions. Recently, [14] extended the theorem to be applicable to multivariate polynomials as well. However, as mentioned earlier, these results do not always hold over the more practical cases of finite word-length bit-vector arithmetic.

The works which come closest to ours have been presented in [24][13]. Both approaches model the given datapath as a polynomial function over a system of finite integer rings. [24] proves equivalence of fixed-size datapaths by using canonical representations of polynomials over finite rings. The concept of vanishing polynomials is used in [13] to derive a symbolic approach to test equivalence of polynomial functions. Both approaches utilize some form of algebraic simplification, which suffers from the well-known intermediate-expression swell problem [25]. In addition, these techniques cannot provide an error trace whenever non-equivalence is detected.

This paper reinterprets the polynomial function model and extends the concepts presented in [8] to derive a novel solution for simulation-based verification of high-level descriptions of arithmetic datapaths. The next two sections cover some preliminary concepts, and then derive the theoretical contributions of this paper. Practical application of our work is subsequently demonstrated.

III. Preliminaries

In what follows, Z corresponds to the set of integers and Z_{2^m} to the finite set of integers $\{0,1,\ldots,2^m-1\}$, over which addition and multiplication are closed. $Z_{2^m}[x]$ denotes the ring of univariate polynomials over the variable x, with coefficients from Z_{2^m}. We use the following multi-index notation: $\mathbf{k} =< k_1,k_2,\ldots,k_d >$ are the (non-negative) degrees corresponding to the d input variables $\mathbf{x} =< x_1,x_2,\ldots,x_d >$, respectively. Also, $n_1,n_2,\ldots,n_d$ and m are the input and output bit-vector sizes. Subsequently, we represent the RTL computations as polyfunctions from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \cdots \times Z_{2^{n_d}}$ to Z_{2^m}, which are defined as [8]:

Definition III.1: A function f from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots \times Z_{2^{n_d}}$ to Z_{2^m} is said to be a polynomial function (or *polyfunction*) if it is represented by a polynomial $F \in Z[x_1,x_2,\ldots,x_d]$; i.e. $f(x_1,x_2,\ldots,x_d) \equiv F(x_1,x_2,\ldots,x_d)$ for all $x_i = 0,1,\ldots,2^{n_i}-1$; $i = 1,2,\ldots,d$. Here, $\equiv$ denotes congruence mod 2^m.

Example III.1: Let $f : Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$ be a polyfunction in two variables (x_1,x_2), defined as:
$f(0,0) = 1$, $f(0,1) = 3$, $f(0,2) = 5$, $f(0,3) = 7$, $f(1,0) = 1$, $f(1,1) = 4$, $f(1,2) = 1$, $f(1,3) = 0$.
Then, f is a polyfunction representable by $F = 1+2x_2+x_1{x_2}^2$, since $f(x_1,x_2) \equiv F(x_1,x_2)\ \%2^3$ for $x_1 = 0,1$ and $x_2 = 0,1,2,3$.

It is possible for a polynomial with non-zero coefficients to *vanish* on such mappings, in which case it represents a *nil polyfunction*. Such polynomials are often called *vanishing polynomials*.

Example III.2: Consider the function $f(x_1,x_2) : Z_2 \times Z_{2^2} \rightarrow Z_{2^3}$ represented by the polynomial $F = 4x_1x_2^2 + 4x_1x_2$. While F has non-zero coefficients, $F\ \%8 \equiv 0, \forall x_1 \in Z_2, \forall x_2 \in Z_4$.

Properties of such polynomials have been extensively studied in number theory and commutative algebra [26] [27] [8]. We summarize some of these results in the context of our work.

A. On Vanishing Polynomials

In the ring Z_{2^m}, let λ denote the least value such that $2^m|\lambda!$. This value is called the *Smarandache function* of 2^m or $SF(2^m)$. For example, $SF(2^3) = 4$ as 8 divides 4! ($= 4 \times 3 \times 2 \times 1 = 8 \times 3$). Note that 8 does not divide 3!, and hence the *least* λ in question $= 4$.

The significance of the above concept can be explained as follows. Consider the ring Z_{2^3}, where $SF(2^3) = 4$ since $2^3|4!$. Consequently, any integer that can be factored into a product of (at least) $\lambda = 4$ consecutive numbers will be divisible by 2^3 and vanish in Z_{2^3}. Now consider a polynomial $f(x)$ in the ring Z_{2^3}, such that $2^3|f(x)$. Therefore, if f, evaluated at x, can be represented as the product of 4 consecutive numbers (depending on x), then f would vanish in Z_{2^3}. A natural example of such a polynomial is: $(x)(x-1)(x-2)(x-3)$. In this regard, Chen [27] proposed a set of monic polynomials, $Y_k(x)$, where each $Y_i(x)$ represents (in polynomial form) a product of i consecutive numbers in x. More formally, we have the following definition and corresponding result:

Definition III.2: Falling factorials of degree k are defined as:
- $Y_0(x) = 1$
- $Y_1(x) = x$

- $Y_2(x) = x(x-1)$

$\vdots$

- $Y_k(x) = x \cdot (x-1) \cdots (x-k+1)$

In the case of d variables, $\mathbf{Y_k}(\mathbf{x}) = \prod_{i=1}^{d} Y_{k_i}(x_i)$.

Lemma III.1: Any polynomial in $Z_{2^m}[x]$ that can be expressed as a factor of $Y_\lambda(x)$ will be divisible by 2^m and vanish.

When such a factorization is not feasible, it is still possible for a given polynomial to $\equiv 0 \ \%2^m$. In this regard, Singmaster [26] identified the constraints on the coefficients which would determine whether the polynomial in question would vanish. We state the following result.

Lemma III.2: The expression $c_k \cdot Y_k(x) \equiv 0$ in $Z_{2^m}[x]$ if and only if $\frac{2^m}{(k!,2^m)} | c_k$;

where,

- c_k is an arbitrary integer in Z,
- $Y_k(x)$ is as defined above,
- $(k!, 2^m)$ is the greatest common divisor (GCD) of $k!$ and 2^m and
- k is the degree of the expression $b \cdot Y_k(x)$, such that $k < \lambda$.

In other words, $c_k \equiv 0 \ \% \frac{2^m}{(k!,2^m)}$ implies that $c_k \cdot Y_k(x) \equiv 0$.

Example III.3: Let $F(x) = 4x^2 - 4x$ over $Z_{2^3}[x]$. Note that $F(x) = 4(x)(x-1) = 4 \cdot Y_2(x)$. Therefore, in this case $k = 2$ and $c_2 = 4$. Also, $c_2 \equiv 0 \% \frac{2^3}{(2!,2^3)} (= 4)$. Because the above condition is satisfied, $F(x) \% 2^3 \equiv 0$. Note if c_2 were replaced by 3, then $F(x) = 3(x)(x-1)$ would not be a vanishing polynomial as $3 \neq 0 \% \frac{2^3}{(2!,2^3)}$.

Based on the above concepts, Singmaster proposed a unique representation for any univariate vanishing polynomial over Z_{2^m}; *i.e.* $F(x) \equiv 0 \ \%2^m$ if and only if it can be represented as:

$$F(x) = Q_\lambda(x) Y_\lambda(x) + \sum_{k=0}^{\lambda-1} c_k Y_k(x) \tag{3}$$

where

- $c_k \in Z$ is a multiple of $\frac{2^m}{(2^m,k!)}$
- $Q_\lambda(x) \in Z$ is any arbitrary polynomial
- $Y_\lambda(x)$ represents the product of λ consecutive numbers.

Example III.4: Let us explain this result using the previous example. Consider $F(x) = 4x^2 - 4x$ over Z_{2^3}. Here, $\lambda = SF(2^3) = 4$. However, F cannot be factored into $Y_4(x)$, therefore $Q_4 = 0$. Similarly, $c_3 = 0$, as F cannot be factored into $Y_3(x)$. Now, F can be factored using $Y_2(x)$, implying $c_2 = 4$ which is a multiple of $\frac{2^3}{(2^3,2!)} = 4$. Therefore, corresponding to Eqn. 3, $F(x) = 4 \cdot Y_2(x) \equiv 0 \ \%2^3$.

The above results were extended by Chen [8] for multivariate polynomials in d variables over $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots \times Z_{2^{n_d}}$ to Z_{2^m}.

B. Multivariate Vanishing Polynomials

The notion of exploiting the value λ for divisibility of polynomials was extended by [8] to expressions in multiple variables. Chen consequently, defines μ_i, which is the minimum of λ and 2^{n_i}, for $1 \leq i \leq d$.

Lemma III.3: If a polynomial $F(\mathbf{x})$ over $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots \times Z_{2^{n_d}}$ to Z_{2^m} can be factorized into a product of μ_i consecutive numbers in *at least one of the variables* x_i, then it vanishes $\%2^m$.

Example III.5: Let $f : Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$ and its corresponding polynomial be $F = x_1^2 x_2 - x_1 x_2$. Here, $\lambda = 4$, degree(x_1) = $k_1 = 2$ and degree(x_2) = $k_2 = 1$. Note that $\mu_1 = min\{2^{n_1}, \lambda\} = min\{2^1, 4\} = 2 = k_1$ (the condition in Lemma III.3 is satisfied) and $\mu_2 = min\{2^{n_2}, \lambda\} = min\{2^2, 4\} = 4 > k_2$. Thus, F can be written as a product of mu_1 consecutive numbers in x_1:

$$\begin{aligned} x_1^2 x_2 - x_1 x_2 &\equiv x_1(x_1 - 1)x_2 \\ &\equiv Y_{<2,1>}(x_1, x_2) \\ &\equiv 0 \ \%2^3 \end{aligned}$$

Lemma III.4: The expression $c_{\mathbf{k}} \cdot \mathbf{Y_k} \equiv 0$ if and only if $\frac{2^m}{(2^m, \prod_{i=1}^{d} k_i!)} | c_{\mathbf{k}}$; where:

- $c_{\mathbf{k}} \in Z$ is an arbitrary integer;
- $\mathbf{k} =< k_1, \ldots, k_d > \in Z^d$ such that $k_i < \mu_i, \forall i = 1, \ldots, d$;
- $\mathbf{Y_k}$ is from Def. III.2 and
- $(2^m, \prod_{i=1}^{d} k_i!)$ is the greatest common divisor (GCD) of 2^m and $\prod_{i=1}^{d} k_i!$.

Similar to the univariate case, this result is equivalent to $c_{\mathbf{k}} \equiv 0 \% \frac{2^m}{(2^m, \prod_{i=1}^{d} k_i!)}$ implies $c_{\mathbf{k}} \cdot \mathbf{Y_k} \equiv 0$.

Example III.6: Consider the polynomial $F = 4x_1 x_2^2 + 4x_1 x_2$ corresponding to $f(x_1, x_2) : Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$. We can use Lemma III.4 to prove that f is a nil polyfunction. Here, $\lambda = 4$; $\mu_1(2) = min\{2, 4\} = 2$, $\mu_2(4) = min\{4, 4\} = 4$. Also, $\mathbf{k} =< k_1, k_2 >=< 1, 2 >$ corresponds to the highest degrees of x_1, x_2. Moreover, $\prod_{i=1}^{2} k_i! = 1! \cdot 2! = 2$.

$$\begin{aligned} F &\equiv 4x_1 x_2^2 + 4x_1 x_2 \\ &\equiv 4 \cdot x_1 \cdot x_2 \cdot (x_2 - 1) \\ &\equiv c_{<1,2>} \cdot Y_{<1,2>}(x_1, x_2) \\ &\equiv 0 \ \%2^3 \end{aligned}$$

because $c_{<1,2>} = 4 \equiv 0 \% \frac{8}{(8,1!\cdot 2!)}$.

The canonical representation for a multivariate vanishing polynomial is shown below. $F(x_1, x_2, \ldots, x_d) \equiv 0 \% 2^m$ if and only if it can be represented as:

$$F(\mathbf{x}) = Q_\mu(\mathbf{x}) Y_\mu(\mathbf{x}) + \sum_{\mathbf{k}} c_{\mathbf{k}} Y_{\mathbf{k}}(\mathbf{x}) \tag{4}$$

where

- $\mu_i = min\{2^{n_i}, \lambda\}$
- $\mu =< \mu_1, \mu_2, \ldots, \mu_d >$
- $Y_\mu(\mathbf{x}) = Y_{\mathbf{k}}(\mathbf{x})$ for some $k_i = \mu_i$
- $c_{\mathbf{k}} \in Z$ is a multiple of $\frac{2^m}{(2^m, \mathbf{k}!)}$ where $\mathbf{k}! = \prod_{i=1}^{d} k_i!$.

Example III.7: Consider a polynomial $F(\mathbf{x}) = x_1^2 + 7x_1 + 4x_1 x_2^2 + 4x_1 x_2$ for $f : Z_2 \times Z_{2^2} \rightarrow Z_{2^3}$. Here, $\lambda = 4$. Further, $\mu_1 = min\{2, \lambda\} = 2$; $\mu_2 = min\{2^2, \lambda\} = 4$. $F(x_1, x_2)$ can be written as follows:

$$\begin{aligned} F(x_1,x_2) &\equiv x_1(x_1-1)+4\cdot x_1\cdot x_2\cdot(x_2-1) \\ &\equiv Y_{<2,0>}(x_1,x_2)+c_{<1,2>}Y_{<1,2>}(x_1,x_2) \\ &\equiv 0\%2^3 \end{aligned}$$

Here, $Y_{<2,0>}(x_1,x_2)$ represents a product of μ_1 consecutive numbers in x_1. Also, $c_{<1,2>}=4$ is a multiple of $8/(8,1!\cdot 2!)=4$. Clearly, F can be written in the form given by Eqn. 4, and is thus a vanishing polynomial.

We now describe how some of these results can be used in the context of this work.

IV. Theory

A. Univariate Polynomials

Given a polynomial function $f(x)$ and its representative polynomial $F(x)$ over Z_{2^m}, we need to reinterpret F in a way that would be more suitable for our purposes. In this context, we first define the forward difference operator (Δ) [28], which is a discrete analog to the derivative of a polynomial function.

Definition IV.1: Let $F(x)$ be a polynomial over Z.

$$\begin{aligned} (\Delta F)(x) &= F(x+1)-F(x) \\ (\Delta^2 F)(x) &= (\Delta F)(x+1)-(\Delta F)(x) \\ &\vdots \\ (\Delta^k F)(x) &= \sum_{i=0}^{k}(-1)^i\binom{k}{i}F(k-i+x) \end{aligned} \quad (5)$$

Let us now apply the forward difference operator on a degree-2 polynomial in x.

Example IV.1: Let $F(x)=4x^2+3x$ be a polynomial in Z. Applying the Δ operator described in Def. IV.1, we get

$$\begin{aligned} (\Delta F)(x) &= F(x+1)-F(x) \\ &= 4(x+1)^2+3(x+1)-(4x^2+3x) \\ &= 8x+7 \\ (\Delta^2 F)(x) &= (\Delta F)(x+1)-(\Delta F)(x) \\ &= 8(x+1)+7-(8x+7) \\ &= 8 \\ (\Delta^3 F)(x) &= (\Delta^2 F)(x+1)-(\Delta^2 F)(x) \\ &= 0 \end{aligned}$$

We now state Newton's interpolation formula [29] based on the above definition. The proof for this formula is widely available in literature and is not reproduced here.

Definition IV.2: If $F(x)$ is a polynomial of degree k with integral coefficients, then it can be written as

$$F(x)=\sum_{i=0}^{k}(\Delta^i F)(0)\binom{x}{i} \quad (6)$$

In other words, the given univariate polynomial can be expressed as an interpolation over a given set of tabulated points, which are in terms of the first value ($F(0)$) and the powers of the forward difference (Δ). It should be noted that the binomial term in Eqn. 6 can be expanded according to,

$$\binom{x}{i}=\frac{x(x-1)\cdots(x-i+1)}{i!}=\frac{Y_i(x)}{i!} \quad (7)$$

The numerator of this term, $Y_i(x)$, is the *falling factorial* from Def. III.2. Eqn. 6 can now be written as:

$$F(x)=\sum_{i=0}^{k}\frac{(\Delta^i F)(0)}{i!}Y_i(x) \quad (8)$$

Since $F(x)$ has integral coefficients, the expression $\frac{(\Delta^i F)(0)}{i!}$ is always an integer for $0\le i\le k$. Note that $F(x)$ has a maximum of $(k+1)$ coefficients. This is illustrated in the following example:

Example IV.2: Consider the polynomial $F(x)=4x^2+3x$ in Z. Here, the degree of $F(x): k=2$ and F can be expanded into $k+1=3$ terms. Using, Eqn. 8 and the values computed in Example IV.1, this function can be expressed as,

$$\begin{aligned} F(x) &= \sum_{i=0}^{2}\frac{(\Delta^i F)(0)}{i!}Y_i(x) \\ &= \frac{(\Delta^0 F)(0)}{0!}\cdot 1+\frac{(\Delta^1 F)(0)}{1!}\cdot x \\ &\quad +\frac{(\Delta^2 F)(0)}{2!}\cdot x(x-1) \\ &= 7x+4x(x-1) \end{aligned}$$

Let us now apply the above interpretation to polynomials over finite integer rings, using the properties of vanishing expressions from Sec. III. Note that Eqns. 5 - 8 still hold, since the coefficients remain integral.

Let F be any given polynomial corresponding to a polyfunction f in Z_{2^m}. Let λ denote the value of $SF(2^m)$. From Eqn. 3, F can be written as,

$$F(x)=Q_\lambda(x)Y_\lambda(x)+R(x) \quad (9)$$

where, *degree* $(R(x))<\lambda$. For any arbitrary $Q_\lambda\in Z[x]$, $Q_\lambda(x)Y_\lambda(x)$ represents a multiple of λ consecutive numbers and is $\equiv 0\ \%2^m$. The degree of $F(x)$ has now been reduced to $<\lambda$. Let us explain this representation with an example.

Example IV.3: Consider the polynomial $F(x)=x^4+3x^3+7x^2+6x$ over Z_{2^3}. The degree of $F(x)$ is 4. We compute $\lambda=SF(2^3)=4$, and divide $F(x)$ by Y_4 to represent it as:

$$F(x)=Y_4(x)+x^3+4x^2+4x \quad (10)$$

Here, $Q_4=1$ and $R(x)=x^3+4x^2+4x$. Now, $Y_4(x)$ represents a product of λ consecutive numbers in Z_{2^3}, and evaluates to $0\%2^3$. Thus, $F(x)=x^3+4x^2+4x$, where the degree is now $3<\lambda$.

Now, using Newton's interpolation formula, we know that any function with integral coefficients can be represented according to Eqn. 8. Hence, we can now rewrite $F(x)$ as

$$\begin{aligned} F(x)=R(x) &= \sum_{k=0}^{\lambda-1}\frac{(\Delta^k R)(0)}{k!}Y_k(x) \\ &= \sum_{k=0}^{\lambda-1}b_k Y_k(x) \end{aligned} \quad (11)$$

since $deg(R(x)) < \lambda$. According to Lemma III.2, if all the coefficients b_k of this expression reduce to 0 when computed $\%\frac{2^m}{(2^m,k!)}$, then $R(x)$ (and correspondingly, $F(x)$) vanishes in Z_{2^m}.

B. Application to Equivalence Verification

We now use the concepts stated in the previous sections to obtain the following result:

Theorem IV.1: Let $F_1(x)$ and $F_2(x)$ be two polynomials with coefficients in Z_{2^m}. To prove $F_1(x)\%2^m \equiv F_2(x)\%2^m$, it is sufficient to show that F_1 and F_2 are equivalent in Z_{2^m} for **any** λ **consecutive** values of x. Here, λ is the least integer such that $2^m|\lambda!$.

Proof:
We use the the results described in previous sections to outline a systematic procedure as part of the proof for Theorem IV.1.

- Compute the value of λ. Express both $F_1(x)$ and $F_2(x)$ in the form of Eqn. 9. The degrees of $R_1(x)$ and $R_2(x)$ are now $< \lambda$.
- Represent $R_1(x)$ and $R_2(x)$ according to Newton's formula, given by Eqn. 11. The coefficients (b_k) are computed for $0 \leq k < \lambda$, and are unique.
- We know that if all the b_k coefficients are $\equiv 0\%\frac{2^m}{(2^m,k!)}$, then the polynomial vanishes. We can use this property to prove equivalence according to:

$$(b_k(F_1) - b_k(F_2)) \equiv 0 \ \%\frac{2^m}{(2^m,k!)} \tag{12}$$

or

$$b_k(F_1) \ \%\frac{2^m}{(2^m,k!)} \equiv b_k(F_2) \ \%\frac{2^m}{(2^m,k!)} \tag{13}$$

Thus, for each pair of corresponding coefficients $b_k(F_1)$ and $b_k(F_2)$, check if they are congruent modulo $\frac{2^m}{(2^m,k!)}$.

- If this check fails, then the polynomials are not equivalent. We halt the procedure. Else, we repeat the above step for all b_k, $0 \leq k < \lambda$.

Following the above procedure, we need to compute and compare the b_k values for *at most* λ steps. Computation of each coefficient $b_k(F_1)$ (or $b_k(F_2)$) requires evaluation of $F_1(x)$ (or $F_2(x)$) according to Eqn. 11. This implies that $F_1(x)$ (or $F_2(x)$) is evaluated a maximum of λ *consecutive* times.

Theorem IV.1 directly follows from this procedure.

Example IV.4: Consider the polynomials $F_1(x) = x^5 + 15x^4 + 5x^3 + x^2 + 2x + 8$ and $F_2(x) = x^4 + 10x^3 + 3x^2 + 2x + 8$ over Z_{2^4}. Here, $\lambda(2^4) = 6$. To check their equivalence, we need to compare the values of $F_1(x)$ and $F_2(x)$ for *any* 6 consecutive values of x.

$$\begin{aligned} F_1(2) &= 8; \quad F_2(2) = 8 \\ F_1(3) &= 0; \quad F_2(3) = 8 \\ F_1(4) &= 0; \quad F_2(4) = 0 \\ F_1(5) &= 0; \quad F_2(5) = 0 \\ F_1(6) &= 0; \quad F_2(6) = 0 \\ F_1(7) &= 0; \quad F_2(7) = 0 \end{aligned}$$

$F_1(3) \neq F_2(3)\%2^4$; hence, these two polynomials are not the same in Z_{2^4}.

C. Multivariate Polynomials

The results of the previous section can be easily extended to arbitrary polynomials in d variables.

Given any polynomial $F(\mathbf{x})$ corresponding to the polyfunction $f : Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}} \rightarrow Z_{2^m}$, we use Eqn. 4 to represent it as

$$F(\mathbf{x}) = Q_\mu(\mathbf{x})Y_\mu(\mathbf{x}) + R(\mathbf{x}) \tag{14}$$

From Lemma III.3, $Q_\mu(\mathbf{x})Y_\mu \equiv 0\%2^m$. This results in reducing the degree $\mathbf{k}$ of $F(\mathbf{x})$, such that $k_i < \mu_i$ for all $1 \leq i \leq d$.

Example IV.5: Let $F(\mathbf{x}) = x_1^2 + 7x_1 + 3x_1x_2^2 + 4x_1x_2$ over $Z_2 \times Z_{2^2} \rightarrow Z_{2^3}$. Here, $\lambda = 4$, and $\mu_1 = min\{2,\lambda\} = 2$ and $\mu_2 = min\{4,\lambda\} = 4$. We represent the polynomial as,

$$F(x_1,x_2) = Y_{<2,0>}(x_1,x_2) + 3x_1x_2^2 + 4x_1x_2 \tag{15}$$

In this case, $Y_{<2,0>}(x_1,x_2)$ is a product of $\mu_1 = 2$ consecutive numbers in x_1 and thus vanishes $\%2^3$. Thus, $F(\mathbf{x}) = R(\mathbf{x}) = 3x_1x_2^2 + 4x_1x_2$ and has a degree $\mathbf{k} =< 1,2 >$, where $k_1 = 1 < \mu_1$ and $k_2 = 2 < \mu_2$.

We now define Newton's interpolation formula for multiple variables.

Definition IV.3: Let F be a polynomial in d variables $x_1,x_2,\ldots,x_d$ with degrees $\mathbf{k} =< k_1,k_2,\ldots,k_d >$. Then, Newton's formula can be written in multi-index notation as,

$$\begin{aligned} F(\mathbf{x}) &= \sum_{\mathbf{i}\leq\mathbf{k}} (\Delta^{\mathbf{i}}F)(\mathbf{0}) \binom{\mathbf{x}}{\mathbf{i}} \\ &= \sum_{\mathbf{i}\leq\mathbf{k}} \frac{(\Delta^{\mathbf{i}}F)(\mathbf{0})}{i_1!\ldots i_d!} \prod_{j=1}^{d} Y_{k_j}(x_j) \\ &= \sum_{\mathbf{i}\leq\mathbf{k}} \frac{(\Delta^{\mathbf{i}}F)(\mathbf{0})}{\mathbf{i}!} Y_{\mathbf{i}}(\mathbf{x}) \end{aligned} \tag{16}$$

In the above, $\mathbf{i} \leq \mathbf{k}$ implies that $i_1 < k_1$, $i_2 < k_2$, ..., $i_d < k_d$. Applying this to $F(\mathbf{x})$, we get

$$\begin{aligned} F(x) = R(\mathbf{x}) &= \sum_{\mathbf{k}\leq\mu} \frac{(\Delta^{\mathbf{k}}R)(\mathbf{0})}{\mathbf{k}!} Y_{\mathbf{k}}(\mathbf{x}) \\ &= \sum_{\mathbf{k}\leq\mu} b_{\mathbf{k}} Y_{\mathbf{k}}(\mathbf{x}) \end{aligned} \tag{17}$$

Again, note that maximum degree of $F(\mathbf{x}) < \mu$. The coefficients $b_{\mathbf{k}}$ of the above formula are computed for the $\mathbf{k} =< k_1,\ldots,k_d >$ vectors, where each $k_i = 0,\ldots,\mu_i - 1$. This corresponds to a maximum of $\prod_{i=1}^{d}\mu_i$ coefficients. We now state the following theorem.

Theorem IV.2: Let $F_1(x_1,\ldots,x_d)$ and $F_2(x_1,\ldots,x_d)$ be two polynomials over $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots Z_{2^{n_d}}$ to Z_{2^m}. To prove/disprove their equivalence, it is sufficient to check for a total of $\prod_{i=1}^{d}\mu_i$ values, where μ_i is defined as the $min\{2^{n_i},\lambda\}$. Note that each 'value' is actually a d-tuple $< x_1,\ldots,x_d >$, such that each x_i corresponds to **any** μ_i **consecutive** values.

Proof:
The proof is based on the corresponding procedure for univariate polynomials, which is extended and reproduced below.

• Compute the values $\mu_1,\ldots,\mu_d$. Express both $F_1(\mathbf{x})$ and $F_2(\mathbf{x})$ in the form of Eqn. 14. The degrees of $R_1(\mathbf{x})$ and $R_2(\mathbf{x})$ are now $< \mu$.
• Represent $R_1(\mathbf{x})$ and $R_2(\mathbf{x})$ according to Newton's formula, given by Eqn. 17. The coefficients ($b_{\mathbf{k}}$) are computed for all $\mathbf{k}$, where $0 \leq k_i < \mu_i$, and are unique.
• We know that if all the $b_{\mathbf{k}}$ coefficients are $\equiv 0\% \frac{2^m}{(2^m,\prod_{i=1}^{d} k_i!)}$, then the polynomial vanishes. As in the univariate case, we use this property to prove equivalence. Thus, for each pair of corresponding coefficients $b_{\mathbf{k}}(F_1)$ and $b_{\mathbf{k}}(F_2)$, check if they are congruent modulo $\frac{2^m}{(2^m,\prod_{i=1}^{d} k_i!)}$.
• If this check fails, then the polynomials are not equivalent. We halt the procedure. Else, we repeat the above step for all $b_{\mathbf{k}}$.

Following the above procedure, we need to compute and compare the $b_{\mathbf{k}}$ values for the tuples $\mathbf{k} =< k_1,k_2,\ldots,k_d >$, where each k_i can take on any μ_i consecutive values. This requires evaluation of $F_1(x)$ (or $F_2(x)$) a maximum of $\prod_{i=1}^{d}\mu_i$ times.

Theorem IV.2 can be inferred from the above procedure.

Example IV.6: Let us now consider the polynomials $F(x_1,x_2) = x_1x_2^3 + 5x_1x_2^2 + 2x_1x_2$ and $F_2 = x_1^4x_2 + 2x_1^3x_2 + 3x_1^2x_2 + x_1x_2^3 + 5x_1x_2^2 + 4x_1x_2$ over $Z_{2^1} \times Z_{2^2} \rightarrow Z_{2^3}$. Here, $\mu_1 = 2$ and $\mu_2 = 4$. We need to check if $F_1 \equiv F_2 \ \ \%2^3$. Therefore, we evaluate both polynomials for a maximum of $\mu_1 \cdot \mu_2 = 8$ tuples. Here, x_1 can take any $\mu_1 = 2$ consecutive values and x_2 can take any $\mu_2 = 4$ consecutive values. Evaluating for $x_1 = 0,1$ and $x_2 = 0,1,2,3$ for Z_{2^3}, we get

$$\begin{aligned}
F_1(x_1=0) = 0 \quad &; \quad F_2(x_1=0) = 0 \\
F_1(x_1=1;x_2=0) = 0 \quad &; \quad F_2(x_1=1;x_2=0) = 0 \\
F_1(x_1=1;x_2=1) = 4 \quad &; \quad F_2(x_1=1;x_2=1) = 4 \\
F_1(x_1=1;x_2=2) = 0 \quad &; \quad F_2(x_1=1;x_2=2) = 0 \\
F_1(x_1=1;x_2=3) = 6 \quad &; \quad F_2(x_1=1;x_2=3) = 6
\end{aligned}$$

Since $F_1(\mathbf{x}) \equiv F_2(\mathbf{x})$ for 2 consecutive values of x_1 and 4 consecutive values of x_2, the two polynomials are equivalent.

V. Results

Using the results from Theorems IV.1 and IV.2, we have been able to perform simulations over a number of designs collected from a variety of benchmark suites.The results are presented in Table I.

The first example is an image rejection computation. The phase-shift keying (PSK) [4] is used in digital communication. The polynomial filters [30] are Volterra models of polynomial signal processing applications. MIBench is a 9^{th}-degree polynomial from [31]. The anti-aliasing function is commonly used in MP3 decoders and is from [4]. Horner polynomials [32] are commonly used in DSP - often implemented using multiply-add-accumulate units. In [4], it was shown how computations by these MAC units can be extracted as polynomials in Horner's form. The last example is a vanishing polynomial of degree 10.

Some of these designs were available as RTL code. The others were available as high-level specifications in MATLAB or C code. RTL code for these reference designs was automatically generated using the MATLAB Simulink and Filter Design toolboxes (particularly for the digital filter designs) [3]. Once the reference RTL descriptions were obtained, they were further optimized using techniques from [4] and [5]. In [4], application of high-level restructuring and symbolic algebra-based transformations was presented for high-level synthesis. These include factorization and expansion, tree-height reduction, etc. The recent work of [5] has derived a sequence of polynomial algebra based transformations to reduce the area-cost of the implementation. This is achieved by modulating and segmenting the coefficients and subsequently removing algebraic redundancy (vanishing polynomials). These transformations were applied to the original RTL description to obtain functionally equivalent implementations.

Subsequently, the data-flow graphs for the original and optimized RTL descriptions were extracted using GAUT [33]. Traversing the DFGs from the inputs to the outputs, the polynomial representations were constructed. The datapath sizes of both inputs and outputs ($n_1,\ldots,n_d$ and m) were also recorded. Using the proposed results, the maximum number of required test vectors was determined for all benchmarks. The descriptions were then simulated with these vectors to verify equivalence. We also wanted to analyze the performance of our approach in the presence of bugs. To verify that our algorithm can detect non-equivalence of designs, we experimented with some designs by arbitrarily changing one or more of the coefficients. In all cases, we were able to detect the erroneous values within the required number of simulations.

A. Limitations of our approach

Our approach works only on arithmetic datapaths whose functionality can be captured as polynomial functions. Hence, we cannot apply our technique to verify gate-level netlists. Also, many DSP systems implement some form of computation approximation, by incorporating various rounding schemes. Our approach is currently restricted inasmuch as it cannot verify those datapaths where intermediate signals have varying precision (due to rounding). Similarly, saturation arithmetic architectures can also not be verified using our technique. Analysis of such designs requires substantially more work, and is the subject of our future investigations.

VI. Conclusions

We have presented a method to determine simulation-bounds for equivalence verification of high-level descriptions of arithmetic datapaths. Our approach models the design as a polyfunction from $Z_{2^{n_1}} \times Z_{2^{n_2}} \times \ldots \times Z_{2^{n_d}} \rightarrow Z_{2^m}$. Established concepts from number theory and commutative algebra have been analyzed and extended to derive the requisite theoretical results. Practical application of these results was also demonstrated for simulation-based verification. We were able to verify equivalence for a number of circuits by simulating the RTL descriptions. Also, the bugs in the design were detected within the proposed bound. As part of future work, we are investigating applications of the proposed results to various other

TABLE I
REQUIRED NUMBER OF SIMULATION VECTORS

Benchmark	Specs	Total	Proposed	Required	EQUIV/
	Var/Deg/$< n_1, \ldots, n_d >/m$	Test Vectors	Test Vectors	Test Vectors	BUG ?
Fault-free circuits					
IRR	$2/4/< 12, 8 >/16$	2^{20}	18^2	18^2	EQUIV
PSK	$2/4/< 11, 14 >/16$	2^{25}	18^2	18^2	EQUIV
Degree-4 filter 1	$3/4/< 15, 11, 13 >/16$	2^{39}	18^3	18^3	EQUIV
Degree-4 filter 2	$1/4/< 12 >/16$	2^{12}	18	18	EQUIV
Savitzky-Golay filter	$5/3/< 16, 16, 14, 12, 8 >/16$	2^{66}	18^5	18^5	EQUIV
4^{th} Order IIR	$2/4/< 24, 29 >/32$	2^{53}	34^2	34^2	EQUIV
MIBENCH	$2/9/< 16, 12 >/16$	2^{28}	18^2	18^2	EQUIV
Faulty circuits					
Anti-alias function	$1/6/< 11 >/16$	2^{11}	18	18	BUG
Horner Polynomial	$3/4/< 10, 8, 16 >/16$	2^{34}	$18^3(5832)$	4335	BUG
Vanishing polynomial	$2/10/< 12, 12 >/16$	2^{24}	$18^2(324)$	103	BUG

datapath computations such as those that implement rounding and saturation arithmetic.

REFERENCES

[1] D. Menard, D Chillet, F Charot, and O. Sentieys, "Automatic Floating-point to Fixed-point Conversion for DSP Code Generation", *in Intl. Conf. Compiler, Architecture, Synthesis Embedded Sys., CASES*, 2002.

[2] I. A. Groute and K. Keane, "M(VH)DL: A MATLAB to VHDL Conversion Toolbox for Digital Control", *in IFAC Symp. on Computer-Aided Control System Design*, Sept. 2000.

[3] MATLAB/Simulink; Matlab to RTL Translator, ", http://www.mathworks.com/products/simulink.

[4] A. Peymandoust and G. DeMicheli, "Application of Symbolic Computer Algebra in High-Level Data-Flow Synthesis", *IEEE Trans. CAD*, vol. 22, pp. 1154–11656, 2003.

[5] S. Gopalakrishnan, P. Kalla, and F. Enescu, "Optimizing Fixed-Size Bit-Vector Arithmetic using Finite-Ring Algebra", *in To appear, IWLS*, 2006.

[6] D. E. Knuth, *The Art of Computer Programming, Vol. II, Seminumerical Algorithms*, Addison Wesley, 1998.

[7] Crandall R. and Pomerance C., *Prime Numbers: a Computational Perspective*, Springer, 2000.

[8] Z. Chen, "On polynomial functions from $Z_{n_1} \times Z_{n_2} \times \cdots \times Z_{n_r}$ to Z_m", *Discrete Math.*, vol. 162, pp. 67–76, 1996.

[9] J. Smith and G. DeMicheli, "Polynomial methods for component maching and verification", *in In Proc. ICCAD*, 1998.

[10] P. Sanchez and S. Dey, "Simulation-Based System-Level Verification using Polynomials", *in High-Level Design Validation & Test Workshop, HLDVT*, 1999.

[11] I. Ugarte and P. Sanchez, "Formal Meaning of Coverage Metrics in Simulation-Based Hardware Design Verification", *in High-Level Design Validation & Test Workshop, HLDVT*, 2005.

[12] N. Shekhar, P. Kalla, Enescu F., and S. Gopalakrishnan, "Exploiting Vanishing Polynomials for Equivalence Verification of Fixed-Size Arithmetic Datapaths", *in Intl. Conf. Computer Design*, 2005.

[13] N. Shekhar, P. Kalla, and Enescu F., "Equivalence Verification Arithmetic Datapaths with Multiple Word-length Operands", *in Design Automation and Test in Europe (DATE)*, 2006.

[14] T. Raudvere, A. K. Singh, I. Sander, and A. Jantsch, "System level verification of digital signal processing applications based on the polynomial abstraction technique", *in Proc. of ICCAD*, pp. 285–290, 2005.

[15] J. E. Eaton, "The Fundamental Theorem of Algebra", *American Mathematical Monthly*, vol. 67, pp. 578–579, 1960.

[16] R. E. Bryant, "A Methodology for Hardware Logic Simulation", *Journal of the ACM*, vol. 38, pp. 299–328, 1991.

[17] R. E. Bryant, "Formal Verification of Memory Circuits by Switch-Level Simulation", *IEEE Transactions on CAD*, vol. 10, pp. 94–102, Jan, 1991.

[18] R. Ernst and J. Bhasker, "Simulation- Based Verification for High-Level Synthesis", *IEEE Design and Test*, vol. 8, pp. 14–20, Jan, 1991.

[19] D. Brand, "Exhaustive Simulation Need Not Require an Exponential Number of Tests", *in Proc. ICCAD*, pp. 98–101, Nov 1992.

[20] E. Clarke, S. German, Y. Lu, H. Veith, and D. Wang, "Executable Protocol Specification in ESL", *in Proc. FMCAD*, Nov 2000.

[21] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling design constraints and biasing in simulation using bdds", *in Proc. ICCAD*, Nov 1999.

[22] K. Shimizu and D. L. Dill, "Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification", *in Proc. DAC*, pp. 801–806, 2002.

[23] K. Shimizu and D. L. Dill, "Using Formal Specifications for Functional Validation of Hardware Designs", *IEEE Design & Test of Computers*, vol. 19, pp. 96–106, 2002.

[24] N. Shekhar, P. Kalla, Enescu F., and S. Gopalakrishnan, "Equivalence Verification of Polynomial Datapaths with Fixed-Size Bit-Vectors using Finit Ring Algebra", *in Intl. Conf. on Computer-Aided Design, ICCAD*, 2005.

[25] J. Moses, "Algebraic simplification: A guide for the perplexed", *Comm. ACM*, vol. 14, pp. 548–560, 1971.

[26] D. Singmaster, "On Polynomial Functions (mod m)", *J. Number Theory*, vol. 6, pp. 345–352, 1974.

[27] Z. Chen, "On polynomial functions from z_n to z_m", *Discrete Math.*, vol. 137, pp. 137–145, 1995.

[28] N. J. A. Sloane and S. Plouffe, *The Encyclopedia of Integer Sequences*, Academic Press, 1995.

[29] C. Jordan, *Calculus of Finite Differences*, New York: Chelsea, 1965.

[30] V. J. Mathews and G. L. Sicuranza, *Polynomial Signal Processing*, Wiley-Interscience, 2000.

[31] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A Free, Commercially Representative Embedded Benchmark Suite", *in IEEE 4th Annual Workshop on Workload Characterization*, Dec 2001.

[32] A. K. Verma and P. Ienne, "Improved use of the Carry-save Representation for the Synthesis of Complex Arithmetic Circuits", *in Proceedings of the International Conference on Computer Aided Design*, 2004.

[33] Universite' de Bretagne Sud LESTER, "Gaut, Architectural Synthesis Tool", *http://lester.univ-ubs.fr:8080*, vol. , 2004.

Design for Verification of the PCI-X Bus

Haja Moinudeen, Ali Habibi and Sofiène Tahar
Electrical and Computer Engineering Department
Concordia University, Montreal, Canada
Email: {haja_m,habibi,tahar}@ece.concordia.ca

***Abstract*— The importance of re-usable Intellectual Properties (IPs) cores is increasing due to the growing complexity of today's system-on-chip and the need for rapid prototyping. In this paper, we provide a design for verification approach of a PCI-X bus model, which is the fastest and latest extension of PCI technologies. We use two different modeling levels, namely UML and AsmL. We integrate the verification within the design phases where we use model checking and model based testing, respectively at the AsmL and SystemC levels. This case study presents an illustration of the integration of formal methods and simulations for the purpose of providing better verification results of SystemC IPs.**

I. Motivation and Proposed Methodology

With the advent of high technology applications, an increasingly evident need has been that of incorporating the traditional microprocessor, memories and peripherals on a single silicon. This is what has marked the beginning of the System-on-Chip (SoC) era. An SoC can be viewed as a collection of various Intellectual Property (IP) cores, with interconnecting buses running among them. There is a dire need for standard buses to connect IPs obtained from different vendors. One such and latest bus standards is PCI-X [8], which is a high performance bus for interconnecting chips, expansion boards, and processor/memory subsystems. It has the performance to feed the most bandwidth-hungry applications and helps to alleviate the I/O bottleneck problem while at the same time maintaining complete hardware and software backward compatibility to previous generations of PCI [8].

In this paper, we present a design for verification effort done for the PCI-X bus. We start with an informal specification of PCI-X and model it with the Unified Modeling Language (UML) in order to have a clear view of the design modules and their interactions. Then, we construct an Abstract Machine Language (AsmL) [3] model from the UML representation.

We define a set of properties of the PCI-X in the Property Specification Language (PSL) [1] that we verify using model checking. Finally, we translate the AsmL model to SystemC [5]. Unfortunately, not all bus properties can be verified using model checking, that is why we use model based testing (MBT) [6] to perform a guided simulation of the IP.

Related work to ours in the context of PCI technologies design and verification environment concerns, in particular, the work of Shimizu *et al.* [7] who presented a specification of the PCI bus as a Verilog monitor. Any modification or refinement of the model provided [7] is complex due to the low level of specification of the bus. Furthermore, the PCI-X standard includes very complex transaction rules in comparison to PCI which cannot be handled only using model checking.

II. PCI-X Bus

PCI-X provides backward compatibility by allowing devices to operate at conventional PCI frequencies and modes. The bus structure includes an arbiter that performs the bus access arbitration among multiple initiators and targets (see Figure 1). Unlike the conventional PCI bus, the arbiter in PCI-X systems monitors the bus in order to ensure good functioning of the bus. PCI-X supports two modes of operations: Mode 1 and Mode 2. In Mode 1 operation, data transfers always use common clock. Mode 2 operation of PCI-X also supports 16 bit bus interface which facilitates low cost interface.

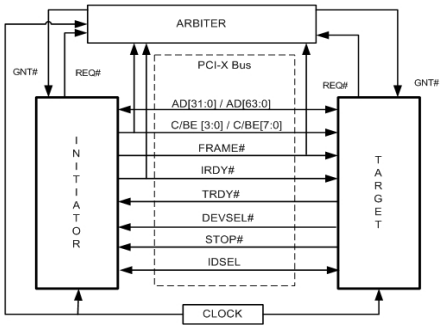


Fig. 1. General Architecture of PCI-X.

III. Design Phases

UML Level: From the specification of PCI-X, we identify the core components of the bus viz, initiators, targets, arbiters, PCI-X bus, which will be represented as classes, where specific instances of the components are called as objects. In addition to these four components, we also added another component, the Simulation Manager (SimManager), in order to have a notion of updates. We modeled different modes and types of operations of PCI-X using sequence diagrams which enable us to model the bus in AsmL easily and efficiently.

AsmL Level: Class diagrams in UML help implementing the classes in AsmL. Each of the five core components of PCI-X has its own data members (signals) and methods (behavior) in addition to a constructor. We also use enumeration features (*enum*) of AsmL to model different modes of PCI-X, different types of transaction phases, the state of the system and the clock. In Figure 2, we show how a target can signal

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

its readiness using the TRDY# signal. We call this method as *PCIX_Target_TRDY_Assert()*. The pre-conditions are the following: TRDY# is *false*, FRAME# and DEVSEL# are *true*, CLK is *CLK_UP*, Phase is *DATA_PHASE_FIRST* and the *AD* of the Bus should be the *ID* of the target. If the pre-conditions are true, then TRDY# will be asserted.

```
public PCIX_Target_TRDY_Assert()
    require me.TRDY = false and Bus.FRAME = true
        and Bus.AD = me.ID and Bus.DEVSEL = true
        and Smanager.CLK = CLK_UP
        and Phase = DATA_PHASE_FIRST
    me.TRDY := true
    me.AD := Bus.AD
    Bus.TRDY := true
    Phase := DATA_PHASE
```

Fig. 2. Target Assert AsmL Method.

SystemC Level: After the AsmL model of a PCI-X is verified against the properties, we translate it to SystemC using a sound syntactical transformation developed by Habibi *et al.* in [4].

IV. Verification Approach

Model Checking: The AsmL model is validated using a set of user-defined PSL properties. Any incorrect property detection halts the reachability algorithms and outputs a trace for counter-examples. Table I provides the results of model checking of PCI-X model with 5 initiators and 5 targets [1]. We show the CPU time, number of states and transitions for the PCI-X model with various properties that we defined. As can be seen from the table, all properties have been verified except Property 6 and Property 7 due to a state explosion problem. For instance, these properties are related to the successful completion of a data transfer which typically takes more cycles than other transactions.

TABLE I
Model Checking Results

Property	CPU Time (s)	States	Transitions
P1	385.24	2169	3250
P2	194.23	1800	2563
P3	150.52	1578	2156
P4	130.45	1489	2096
P5	156.35	1478	2265
P6	–	–	–
P7	–	–	–
P8	173.50	1925	2439
P9	174.47	2013	2698
P10	178.42	1873	2359
P11	256.63	2192	2980
P12	143.52	1356	1923

Model Based Testing: In MBT the behavior of a system is defined in terms of actions that change the state of the system. Such a model of the system results in a well-defined Finite State Machine (FSM) which helps understand and predict the system's behavior. We first generate the FSM of the PCI-X model in AsmL based on the algorithm developed by Grieskamp *et al.* in [2]. Then, using existing graph traversing techniques, test cases are obtained from the generated FSMs to validate the PCI-X model. Table II shows the CPU time, states and number of transitions in the generated FSM, for several combinations of initiators and targets. Using the generated FSM, we apply various techniques to choose the tests. We took advantage of several efficient graph traversing techniques in the open literature, in particular, the Chinese Postman Tour (CPT) and Random walk methods. MBT was of a great help in identifying several bugs in the SystemC PCI-X model we developed.

[1]Experimentation platform: Pentium IV(2.4 GHz) / 768 MB of memory.

TABLE II
FSM Generation: Direct Algorithm.

Number of		CPU	States	Transitions
Initiators	Targets	Time (s)		
1	5	27.95	234	253
2	5	59.50	466	505
3	5	108.04	698	757
4	5	171.73	930	1009
5	2	69.89	472	511
5	3	118.20	702	761
5	4	204.93	932	1011
5	5	254.82	1162	1261
10	10	2925.31	4622	5021

V. Conclusion

We presented a design for verification approach applied on the latest high speed PCI-X standard bus. Starting with a UML formal specification, we derived an AsmL model which we model checked against a set of PSL properties. We then translated the AsmL model to SystemC on which we investigated the potential of model based testing approach for SystemC designs. The traversal of the FSM was performed to generate test cases. The final PCI-X SystemC IP was thoroughly and formally verified, which makes it very suitable for use as external monitor to validate existent PCI-X compatible IPs. We believe that our approach shows how one can improve the verification of SystemC models by integrating formal methods and guided simulation in a single design flow.

References

[1] Accellera Organization. Accellera Property Specification Language Reference Manual, version 1.1. www.accellera.org, 2005.

[2] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. *Software Engineering Notes*, 27(4):112–122, 2002.

[3] Y. Gurevich, B. Rossman, and W. Schulte. Semantic Essence of AsmL. Technical report, Microsoft Research, MSR-TR-2004-27, March 2004.

[4] A. Habibi and S. Tahar. On the Transformation of SystemC to AsmL using Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 131:39–49, May 2005.

[5] Open SystemC Initiative. www.systemc.org, 2006.

[6] H. Robinson. Model-based testing. website: http://www.geocities.com/model_based_testing/, 2006.

[7] K. Shimizu, D. L. Dill, and A. J. Hu. Monitor-based formal specification of PCI. In *Formal Methods in Computer-Aided Design*, pages 335–353. LNCS 1954, Springer-Verlag, 2000.

[8] PCI Special Interest Group. Website: www.pcisig.com, 2006.

Formal Analysis and Verification of an OFDM Modem Design using HOL

Abu Nasser M. Abdullah*, Behzad Akbarpour§ and Sofiène Tahar*
*Department of ECE, Concordia University, Montreal, QC, H3G 1M8, Canada
Email: {m_nasser, tahar}@ece.concordia.ca
§ Computer Laboratory, University of Cambridge, Cambridge, CB3 0FD, UK
Email: Behzad.Akbarpour@cl.cam.ac.uk

***Abstract*— In this paper we formally specify and verify an implementation of the IEEE802.11a standard physical layer based OFDM (Orthogonal Frequency Division Multiplexing) modem using the HOL (Higher Order Logic) theorem prover. The versatile expressive power of HOL helped model the original design at all abstraction levels starting from a floating-point model to the fixed-point design and then synthesized and implemented in FPGA technology. The paper also investigates the rounding error accumulated during ideal real to floating-point and fixed-point transitions at the algorithmic level.**

I. Introduction

OFDM [7] is a modulation technique where data is spread over many channels and transmitted in parallel. In this method, an available bandwidth is divided into several subchannels which are independently modulated with different carrier frequencies. The name orthogonal comes from the fact that the subcarriers are orthogonal to each other. This orthogonality eliminates the need of guard band and the carriers can be placed very close to each other without causing interference and thus conserving bandwidth. Due to these characteristics of OFDM, it is used in many applications such as digital audio broadcasting and IEEE802.11a/g wireless LAN standard.

In this paper, we use theorem proving techniques based on the HOL system [2] to verify an implementation of an OFDM modem for the physical layer of the IEEE802.11a standard [6]. The specifications and implementations of the design blocks are modeled in formal logic and then mathematical theorems are proved for their correctness. Besides, we carry out a formal error analysis of the OFDM modem in order to analyze the round-off error accumulation while converting from one number domain to the other. They are a direct application of a general methodology proposed in [1] for the formal modeling and verification of DSP (Digital Signal Processing) designs. The results of this paper demonstrate the functional correctness of the OFDM system and proves the feasibility of applying formal methods for similar systems.

There exists a couple of work related to the application of formal methods for the IEEE802.11 using probabilistic model checking technique based on the PRISM tool [5, 8]. They are dealing with the protocol verification and address the verification issues related to the upper layers of the OSI model. In contrast, in this paper we concentrate on the physical layer and its hardware implementation. Moreover, instead of model checking with its inherent state space limitations, we use theorem proving based on HOL.

Previous work on formal error analysis was done by a number of researchers including Harrison [3], Huhn *et al.* [4], and Akbarpour *et al.* [1]. In particular the work in [1] proposed an error analysis technique in HOL for the transition from ideal real to floating- and fixed-point levels. In this paper, we intend to extend this work using a larger case study (OFDM modem).

II. OFDM Modem and Verification Methodology

A standard block diagram implementation of the IEEE802.11a OFDM modem is shown in Figure 1. The design is implemented in Xilinx Virtex II FPGA. The main RTL blocks are the quadrature amplitude modulation (QAM), the demodulation (DQAM), the serial to parallel (S/P), parallel to serial (P/S), and the guard interval insertion and removal blocks. The core computational blocks are fast Fourier transform (FFT) and inverse fast Fourier transform (IFFT). The OFDM modem design was first modeled in the floating-point domain. The second step in the design flow was fixed-point modeling and simulation. Then developed design blocks are implemented in VHDL. Finally, the RTL design is synthesized and mapped into FPGA.

The formal specification, verification and error analysis used in this paper is adopted from the DSP verification framework proposed by Akbarpour *et al.* [1] (see Figure 2). Thereafter, the ideal real specification of the OFDM modem algorithms and the corresponding floating-point (FP) and fixed-point (FXP) designs, as well as the RTL implementation are all provided in higher-order logic based on the idea of shallow embedding of languages. For the transition from real to FP and FXP levels, an error analysis is used in which the real values of the floating-point and fixed-point outputs are compared with the corresponding output of the ideal real

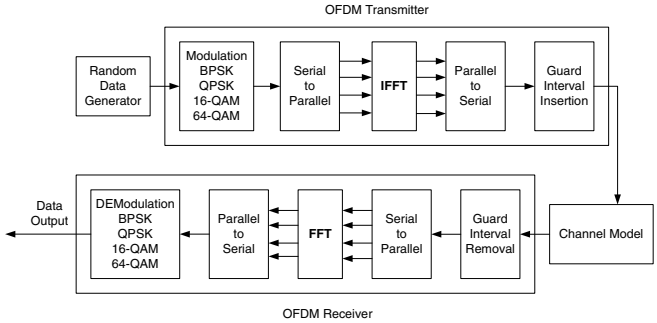


Fig. 1. OFDM Block Diagram [6]

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

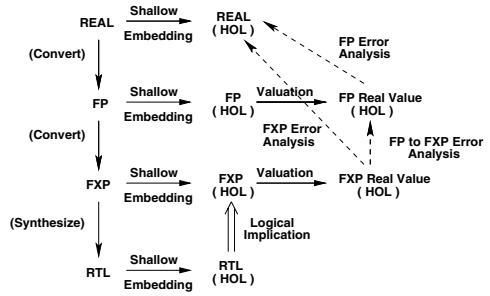


Fig. 2. DSP Specification and Verification Approach [1]

specification. The verification of the RTL is performed using well-known classical hierarchical proof approaches in HOL.

III. Verification of RTL Blocks

For the formal verification of the RTL blocks, we first model the QAM, DQAM, S/P and P/S blocks using higher-order logic. Then a specification of the design is selected either from the IEEE802.11a standard or from existing generic behavioral models of S/P and P/S. Having both the specification and implementation embedded in HOL, we set a relationship between them as a mathematical theorem. We have used the HOL theories *wordTheory* and *realTheory* to build many helpful definitions and lemmas to prove the above theorems and thus established the correctness of the RTL blocks formally.

The main purpose for using formal verification was to find bugs in the design. We did not find any major bugs except in one case. Namely, for the QAM block, it is given in the IEEE802.11a standard that the input for a 64-QAM modulation must follow a specific constellation diagram. The constellation gives output between -7 to 7, where the x and y axis are labeled as -1, -3, -5, -7, 1, 3, 5, 7—in total eight values, so, three bits should have been enough. But, the implementation used 16 bit 2's complement to represent these eight numbers. If the IEEE802.11a standard is to be followed exactly, then this issue might have resulted in a bug in the design; but the standard gives some flexibility to the designers and thus the design is implemented in such a way that the output from the modulation block reaches the IFFT block through S/P block with high resolution in order to get more precise computation after applying fast fourier transform. We encoded the specification in HOL as stated by the standard and same encoding is carried out for the implementation without delving into the designer's point of view. As, we were aware about the deviation in the implementation at the time of verification, we constrained it using the proper number of bits. The same comments are applied to the DQAM block. For the rest of the blocks, we did not find any issue like this.

IV. Error Analysis

In the error analysis of the OFDM modem, we focus on the two computational blocks, FFT and IFFT. Both blocks are probably the most widely used DSP cores, which do introduce computation errors and considered as *raison d'être* of OFDM system. We use HOL to model the computational blocks and the accumulated errors due to the conversion from one domain to the next using different established theories and lemmas built in HOL. To accomplish the complete theory of error analysis, we proved three main theorems based on formalized real and imaginary parts of the combined FFT-IFFT expansion and also theorems related to the error for arithmetic operations. All definitions were derived from many existing HOL theories, e.g., *realTheory*, *boolTheory*, *ieeeTheory*, *wordTheory*, etc. In particular, we used the *floatTheory* and *fxpTheory* to overload the operators for establishing the real, floating-point and fixed-point counterparts of the design.

Given the nature of the formal specification and proofs conducted for the above error analysis, the use of higher-order logic was imperative. No design flaws have been found through this error analysis as the implementation of the OFDM modem strictly adhered to the error boundaries provided in the IEEE802.11a standard specification.

V. Conclusion

This paper is mainly an application of formal verification techniques, in a new domain—an implementation of an OFDM modem based on the IEEE standard. The OFDM design is fairly complex. Its verification in HOL took about 6 months to complete for a learned HOL user. We formally modeled and verified the RTL blocks against the corresponding specifications in the IEEE802.11a standard. We also analyzed the errors in the OFDM system occurring at the time of converting from one number domain to the other, for all three domains—ideal real, floating-point, and fixed-point numbers. We used the IFFT-FFT combination as a model for the error analysis of the whole system. Then we derived fundamental theorems for the accumulation of round-off error in the OFDM system. This formalization can be considered as a large application of the formal error analysis using HOL theorem proving. As a future work, we will look at a hybrid verification approach linking HOL to some automated computer algebra tools such as Maple or Mathematica. The goal is to achieve more efficient verification process.

References

[1] B. Akbarpour and S. Tahar. A Methodology for the Formal Verification of FFT Algorithms in HOL. In *Formal Methods in Computer-Aided Design*, LNCS 3312, pages 37–51. Springer-Verlag, 2004.

[2] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[3] J. Harrison. Floating Point Verification in HOL Light: the Exponential Function. Technical Report 428, University of Cambridge Computer Laboratory, Cambridge, UK, 1997.

[4] M. Huhn, K. Schneider, T. Kropf, and G. Logothetis. Verifying Imprecisely Working Arithmetic Circuits. In *Design Automation and Test in Europe*, pages 65–69, Munich, Germany, March 1999.

[5] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic Model Checking of the IEEE802.11 Wireless Local Area Network Protocol. In *PAPM/PROBMIV*, LNCS 2399, pages 169–187. Springer-Verlag, 2002.

[6] F. Manavi. Implementation of OFDM Modem for the Physical Layer of IEEE802.11a Standard Based on XILINX VIRTEX-II FPGA. Master's thesis, Dept. of ECE, Concordia University, Montreal, QC, Canada, 2004.

[7] R. V. Nee and R. Prasad. *OFDM for Wireless Multimedia Communications*. Artech House Publishers, 2000.

[8] A. Roy and K. Gopinath. Improved Probabilistic Models for 802.11 Protocol Verification. In *Computer Aided Verification*, LNCS 3576, pages 239–252. Springer-Verlag, 2005.

A Formal Model of Lower System Layers

Julien Schmaltz[1]
Saarland University, Saarbrücken, Germany
Email: julien@cs.uni-sb.de

***Abstract*— We present a formal model of the bit transmission between registers with arbitrary clock periods. Our model considers precise timing parameters, as well as metastability. We formally define the behavior of registers over time. From that definition, we prove, under certain conditions, that data are properly transmitted. We discuss how to incorporate the model in a purely digital model. The hypotheses of our main theorem define conditions that must be satisfied by the purely digital part of the system to preserve correctness.**

I. Introduction

Embedded systems rely on different *layers*. A particular application is designed and compiled to be executed by an real time operating system on *several* distributed processors. The *pervasive* verification of such artifacts is achieved by the proof of a theorem stating that the distributed hardware simulates the initial application. This proof must show the formal correctness of safety critical applications, processors, real time operating systems, compilers and communication systems. In the context of automotive systems, a pencil and paper proof of an entire system has already been sketched [3]. In this paper, we provide, for part of this proof, a slightly more general theory, which has been implemented in an automated theorem prover for higher-order logics.

We focus on the very lowest layer of embedded communications. A sender loads a register at some clock rate clk_s, with clock period τ_s. On the other side, a receiver samples the output value of the sender register at some clock rate clk_r, with clock period τ_s (See Fig. 1). Clock periods are constant over time. We model clocks as offset/period pairs. Functions α and τ access clock components. The *date* of clock edge c equals the product of c with the clock period, plus some offset: $e(c, clk) = \alpha(clk) + c \cdot \tau(clk)$. The date of the c^{th} rising of clock clk_u of unit u is noted $e_{clk_u}(c)$. To simplify our notations, we shall write $e_u(c) = \alpha_u + c \cdot \tau_u$. Signals are functions from time to a three valued logic: 1 and 0 for "high" and "low" voltage and Ω for any voltage.

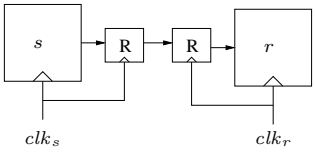


Fig. 1. Communication between registers.

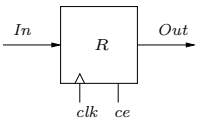


Fig. 2. Register.

[1]A more detailed presentation of this work is available at *http://www-wjp.cs.uni-sb.de/leute/private_homepages/julien/.*

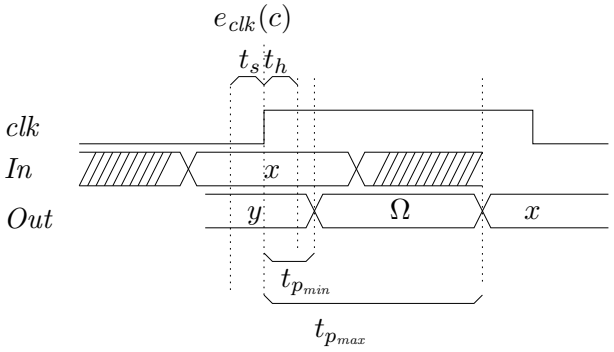


Fig. 3. Behavior of the register w.r.t. edge c

II. Registers

Registers (see Fig.2) have one input In, one clock clk, one control signal ce and one output Out. In Fig. 3, we detail the behavior of a register. If ce is low, the register keeps its old value (at the previous cycle $c-1$); if ce is high the output keeps its previous value during $t_{p_{min}}$, then is undefined (*i.e.* is Ω) to finally reach its final output at time $e_{clk}(c) + t_{p_{max}}$.

To sample the input value properly, signals In and ce must be *defined* and *stable* during the setup time (noted t_s) and the holding time (noted t_h). The *metastability window* w.r.t. some edge c is defined by interval $[e_{clk}(c) - t_s : e_{clk}(c) + t_h]$. A signal is *defined* if it holds a *Boolean* value during some time interval. A signal is stable if it keeps the same value during some time interval. These two conditions are defined as follows:

$$stadep(t_1, t_2, s) \triangleq \exists b \in \mathbb{B}, \forall t \in [t_1 : t_2], s(t) = b$$

If ce or In do not satisfy $stadep$ during the metastability window, the register may get *metastable* and cease for a while to act as a digital device. During that time its output is undefined.

The formal definition of this behavior is given by function R below. We need to consider both cycles and times. Function R represents the state of register R for all times t during cycle c. In other words, we represent the output value for all times during the c^{th} cycle after some "reset". To make it total, Ω is output for t's that are outside the cycle.

Definition 1: **Register.**
$R(c, clk, ce, In, Out^0) \triangleq$
if $c = 0$ **then** Out^0 **else**
if $stadep(e_{clk}(c) - t_s, e_{clk}(c) + t_h, ce)$
$\wedge stadep(e_{clk}(c) - t_s, e_{clk}(c) + t_h, In)$ **then**

Proceedings of the Formal Methods in Computer Aided Design (FMCAD'06)
0-7695-2707-8/06 $20.00 © 2006 IEEE

if $ce(e_{clk}(c)) = 1$ **then**

$\lambda t.$ { **if** $t \in e_{clk}(c) + (0 : t_{p_{min}}]$ **then** $R(c-1, clk, ce, In, Out^0)(e_{clk}(c))$
if $t \in e_{clk}(c) + (t_{p_{min}} : t_{p_{max}})$ **then** Ω
if $t \in [e_{clk}(c) + t_{p_{max}} : e_{clk}(c+1)]$ **then** $In(e_{clk}(c))$
if $t \notin (e_{clk}(c) : e_{clk}(c+1)]$ **then** Ω

else ;; keep old value

$\lambda t.$ { **if** $t \in (e_{clk}(c) : e_{clk}(c+1)]$ **then** $R(c-1, clk, ce, In, Out^0)(e_{clk}(c))$
else Ω

endif

else ;; metastability

$\lambda t.\Omega$

endif

endif

III. Connecting Registers

A sender units loads register R_s, the output of which is read by register R_r. The value output by R_s at cycle c is "seen" by register R_r at the first edge after the minimum propagation delay. Let $cy(c)$ be that edge and defined as follows:

$$cy(c) = Min\{c' | e_r(c') + t_h > e_s(c) + t_{p_{min}}\} \qquad (1)$$

To make sure that most of the time, the receiver will not sample during the metastability window, the output of the sender is kept constant for several cycles (say k cycles). Consequently, there is only one metastability window and if k is big enough there exists a "safe sampling window" (noted SSW, and defined by interval $(e_s(c) + t_{p_{max}} : e_s(c+k+1) + t_{p_{min}}]$) in which the receiver can sample properly. To be robust w.r.t. alterations of the transmitted bits, the receiver performs a majority vote over n bits. Thus, SSW must be large enough to entail n receivers cycles outside the metastability window. By definition, $cy(c)$ is in an interval not larger than a receiver period, *i.e.* $cy(c) \in e_s(c) + t_{p_{min}} + (0 : \tau_r]$. From this upper bound, we can derive the lower bound of k:

$$BigEnough(k, n) \equiv \tau_s \cdot (k+1) \geq \tau_r \cdot (n+2) \qquad (2)$$

If SSW is big enough, it entails n receiver cycles.

Theorem 1: **S.S.W. Long Enough.**

$$BigEnough(k, n) \rightarrow \forall l \in [0 : n], e_r(cy(c) + 1 + l) \in \text{SSW}$$

Proof: By induction on n. Because $\tau_r > t_{p_{min}}$, the cycle following $cy(c)$ is in SSW. This concludes the base case. The induction hypothesis leaves us with cycle $cy(c)+n+1$, which is in SSW by definition of $BigEnough(k, n+1)$. ■

If the sender creates an SSW large enough (the first 7 hypotheses of the theorem below), the receiver will recover n times the value sent at (sender) cycle c.

Theorem 2: **Correct Transfer.**

$$\begin{aligned}
& BigEnough(k, n) \wedge ce_s(e_s(c)) = 1 \wedge c > 0 \\
\wedge\ & stadep(e_s(c) - t_s, e_s(c) + t_h, ce_s) \\
\wedge\ & stadep(e_s(c) - t_s, e_s(c) + t_h, In_s) \\
\wedge\ & \forall l \in [1 : k+1], \\
& stadep(e_s(c+l) - t_s, e_s(c+l) + t_h, In_s) \\
\wedge\ & \forall l \in [1 : k+1], \\
& stadep(e_s(c+l) - t_s, e_s(c+l) + t_h, ce_s))) \\
\wedge\ & \forall l \in [1 : k], ce_s(e_s(c+l) = 0 \wedge \forall t, ce_r(t) = 1 \\
\wedge\ & \forall c, In_r = R(c, clk_s, ce_s, In_s, Out_s^0) \\
\rightarrow\ & \forall l \in [0 : n],
\end{aligned}$$

$$R(cy(c) + 1 + l, clk_r, ce_r, In_r, Out_r^0)(e_r(cy(c) + 2 + l)) = In_s(e_s(c))$$

Proof: From the first hypothesis and Theorem 1, we have n cycles in SSW. The first and the last cycles of SSW are proved by definition of R; the rest by induction on k. ■

IV. Connection with a Fully Digital World

Our goal is to integrate our theory in existing correctness proofs, which consider "untimed" registers (* denotes bit lists):

Definition 2: **Untimed Register.**

$\widehat{R}(c, ce^*, In^*, Out^0) \triangleq$
if $c = 0$ **then** Out^0 **else if** $hd(ce^*) = 1$ **then** $hd(In^*)$
else $\widehat{R}(c-1, tl(ce^*), tl(In^*), Out^0)$ **endif endif**

Because of metastability, the connection with function R is not direct. Let f generate a signal $s(t) = f_{clk}(l)$ from a clock and a bit list, such that for all times t around edge $i+1$, $s(t)$ equals the i^{th} bit of In_l. This is formally expressed by $\forall t, i, t \in e_{clk}(i+1) + [-t_s : t_h] \rightarrow f(l, clk) = l[i]$. Let Dig be such that $Dig(0) = 0$, $Dig(1) = 1$, and $Dig(\Omega) = x \in \{0, 1\}$.

Theorem 3: **Connection Theorem.**

$$\widehat{R}(c, ce^*, In^*, Out^0) = Dig(R(c, clk, f_{clk}(ce^*), f_{clk}(In^*), Out^0)(e_{clk}(c+1)))$$

Proof: By induction on c. ■

To instantiate Theorem 2, one only needs to prove that the digital model satisfies the hypotheses. At each cycle, digital signals have a defined and stable value. The only remaining property is that the sender register is kept stable long enough.

V. Conclusion

We have embedded our theory in Isabelle [2]. Currently, time is defined over the naturals. The dense property of the reals is not required. The next affected cycle is not defined as a minimum. Conclusions of theorems are weaken by supposing the existence of a receiver cycle in $e_s(c) + t_{p_{min}} + (0 : \tau_r]$.

Because of its very low level, our model can serve as a justification for making idealize timing diagrams for system component specification as in data sheets. Another research direction is to justify abstract models (*e.g.* [1]) from our theory.

References

[1] G.M. Brown and L. Pike, Easy Parameterized Verification of Biphase Mark and 8N1 Protocols, TACAS'06, Springer, 2006.

[2] T. Nipkow, L.C. Paulson and M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002.

[3] S. Beyer *et al.*, Towards the Formal Verification of Lower System Layers in Automotive Systems, ICCD 2005.

Author Index: FMCAD 2006

Formal Methods in Computer Aided Design